Вінницький національний технічний університет
Факультет інформаційних технологій та комп'ютерної інженерії
Кафедра програмного забезпечення

Пояснювальна записка

до магістерської кваліфікаційної роботи
магістр
(ступінь вищої освіти)

на тему: Розробка методів та засобів побудови для моніторингу запитів в додатках на основі архітектури мікросервісів

Студента 2 курсу, групи 1ПІ-18м (д/ф)
Спеціальності
121 «Інженерія програмного забезпечення»
Біліченка І.Ю.
(прізвище та ініціали)
Керівник к.т.н., доц. Ракитянська Г.Б.
(посада, вчене звання, науковий ступінь, прізвище та ініціали)

Рецензент: к.т.н., доц. каф. КН Крилик Л.В.
(прізвище та ініціали)

ВНТУ– 2019

Вінницький національний технічний університет
Факультет інформаційних технологій та комп'ютерної інженерії
Кафедра програмного забезпечення
Ступінь вищої освіти – магістр
Спеціальність 121 – Інженерія програмного забезпечення

ЗАТВЕРДЖУЮ
Завідувач
кафедри ПЗ
Романюк О. Н.
"___"_____20__року

З А В Д А Н Н Я
НА МАГІСТЕРСЬКУ КВАЛІФІКАЦІЙНУ РОБОТУ
СТУДЕНТУ
Біліченку Іллі Юрійовичу

1. Тема роботи: «Розробка методів та засобів побудови для моніторингу запитів в додатках на основі архітектури мікросервісів»
керівник роботи: к.т.н., доцент кафедри ПЗ Ракитянська Г.Б. затверджені наказом вищого навчального закладу від "___"_____20__року №___

2. Строк подання студентом роботи _____

3. Вихідні дані до роботи :
Мова програмування: TypeScript
Технологія розробки: VS Code Studio
Браузери (або ОС):
ChromeAndroid >= 63,
Chrome >= 56,
Edge >= 16,
Firefox >= 57,
Safari >= 11

4.Зміст розрахунково-пояснювальної записки (перелік питань, які потрібно розробити): вступ; аналіз предметної галузі розподіленого трасування запитів ; розробка моделей і методів для моніторингу запитів; програмна реалізація

додатку; тестування розробленого програмного продукту; економічна частина; висновки; перелік посилань; додатки.

5. Перелік графічного матеріалу (з точним зазначенням обов'язкових креслень): мета роботи, об'єкт та предмет дослідження; основні задачі дослідження; наукова новизна, практичне значення; архітектура системи; приклад вигляду розробленої програми; висновки.

6. Консультанти розділів магістерської кваліфікаційної роботи

Розділ	Прізвище, ініціали та посада консультанта	Підпис, дата	
		завдання видано	завдання прийнято
1–4	к.т.н. Ракитянська Г.Б., доцент кафедри ПЗ		
5	к.е.н. Бальзан М. В., доцент кафедри ЕПВМ		

7. Дата видачі завдання_____

КАЛЕНДАРНИЙ ПЛАН

№ з/п	Назва етапів магістерської кваліфікаційної роботи	Строк виконання етапів роботи	Примітка
	Техніко-економічне обґрунтування доцільності розробки сервісу для моніторингу запитів в додатку на основі архітектури мікросервісів		Вик.
	Розробка моделей і методів для моніторингу запитів		Вик.
	Програмна реалізація додатку для моніторингу запитів		Вик.
	Тестування роботи додатку		Вик.
	Економічне обґрунтування розробки програмного продукту		Вик.
	Оформлення матеріалів до захисту МКР		Вик.

Студент_____ **Біліченко І.Ю.**

(підпис) (прізвище та ініціали)

Керівник магістерської кваліфікаційної роботи_____ **Ракитянська Г.Б.**

(підпис) (прізвище та ініціали)

АНОТАЦІЯ

Магістерська робота присвячена розробці методів та засобів для моніторингу запитів в додатках на основі мікросервісів. Було оглянуто існуючі рішення, на основі яких було сформовано вимоги до програмного модуля. Створений програмний додаток написаний на мові програмування TypeScript, програмне забезпечення характеризується зручністю та зрозумілістю інтерфейсу, швидкістю та точністю опрацювання даних, що забезпечує всі вимоги користувача щодо організації процесу.

ABSTRACT

The Master's thesis is dedicated to the development of a service for monitoring requests in an application based on the architecture of microservices. Existing solutions were reviewed on the basis of which the requirements for the software module were formed. Created a software application written in TypeScript programming language, the software is characterized by the convenience and clarity of the interface, the speed and accuracy of processing data, which provides all the requirements of the user.

ЗМІСТ

ВСТУП 7
1 АНАЛІЗ ПРЕДМЕТНОЇ ГАЛУЗІ РОЗПОДІЛЕНОГО ТРАСУВАННЯ ЗАПИТІВ 11
1.1 Аналіз предметної галузі 11
1.2 Вибір моделі життєвого циклу для розробки додатку 14
1.3 Характеристика і аналіз аналогів 18
1.4 Проблеми які можна вирішити з підтримкою трасування 19
1.5 Постановка задачі 20
1.6 Висновок 21
2 РОЗРОБКА МОДЕЛЕЙ І МЕТОДІВ ДЛЯ МОНІТОРИНГУ ЗАПИТІВ 22
2.1 Розробка серверної частини 22
2.2 Розробка візуальної клієнтської частини 26
2.3 Метод додавання власних ключів та значень в заголовок запиту при розподіленому трасуванні 34
2.4 Метод сортування запитів із додаванні фільтрів з розширеними умовами пошуку 36
2.5 Висновок 38
3 ПРОГРАМНА РЕАЛІЗАЦІЯ ДОДАТКУ ДЛЯ МОНІТОРИНГУ ЗАПИТІВ 39
3.1 Обґрунтування вибору мови програмування 39
3.2 Обґрунтування вибору середовища розробки 43
3.3 Програмна реалізація 46
3.3 Висновок 49
4 ТЕСТУВАННЯ РОЗРОБЛЕНОГО ПРОГРАМНОГО ПРОДУКТУ 50
4.1 Тестування додатку 50
4.2 Тестовий приклад роботи програмних модулів та аналіз результатів 56
4.3 Висновок 58
5 ЕКОНОМІЧНА ЧАСТИНА 59
5.1 Оцінювання комерційного потенціалу розробки 59
5.2 Прогнозування витрат на виконання науково-дослідної роботи та конструкторсько– технологічної роботи. 60
5.3 Прогнозування комерційних ефектів від реалізації результатів розробки. 63
ВИСНОВКИ 69
ПЕРЕЛІК ПОСИЛАНЬ 70
ДОДАТКИ 72

Додаток А. Технічне завдання 73
Додаток Б. Програмний код додатку 76
Додаток В. Графічні матеріали 89

ВСТУП

Забезпечення стабільної узгодженої роботи кожного сервісу є одним з факторів, що визначають стабільність і якість роботи додатку. Розподілене трасування, яке також називається розподіленим відстеженням запитів, - це метод для IT та команд DevOps для моніторингу додатків, особливо тих, що складаються з мікросервісів. Розподілене трасування допомагає точно визначити, де трапляються збої та що викликає неоптимальну ефективність. Розподілене трасування вимагає від розробників програмного забезпечення додавати інструментарій до коду програми. Цей інструментарій надає інформацію, щоб адміністратор міг аналізувати продуктивність, а розробник може налагоджувати операції складного програмного забезпечення.

Відстеження запитів - це фундаментальна практика в розробці програмного забезпечення. Розробники використовують невеликі частини інструментального коду, створені для відстеження та доставки відповідних показників про масив важливих способів поведінки в коді програми. У більшості випадків мета полягає у відстеженні поведінки запитів, таких як запити користувачів на веб-сайт, що відбуваються в рамках програмного забезпечення. Це поширене використання - це те, що дало цю методику відстеження запиту на ім'я.

Відстеження запитів стикається з серйозними проблемами при використанні в архітектурах розподіленого програмного забезпечення з декількома функціональними модулями або послугами. Служби масштабують незалежних, внаслідок чого багато ітерацій однієї функції, що потенційно працює в різних хост-системах або навіть різних середовищах. Хоча простежити запит через функцію X у монолітній програмі - це проста справа, архітектура на основі мікросервісів може виконати 20 ітерацій функції X на чотирьох серверах у двох центрах обробки даних. Ці служби повинні працювати разом та взаємодіяти з іншими службами, щоб формувати повну заявку. Відстеження запитів повинно продовжувати збирати показники, оскільки запити проходять через безліч різних служб і декількох примірників послуг.

При вирішенні поставлених завдань буде застосовано методи моделювання системи на основі технології SADT, метод проектування бази даних «сутність-зв'язок», а також засоби програмування клієнтської частини автоматизованої системи, засновані на технології Casandra DB, що надає можливість звернення до бази даних.

Зв’язок роботи з науковими програмами, планами, темами. Робота виконувалась згідно плану виконання наукових досліджень на кафедрі програмного забезпечення.

Практична цінність отриманих результатів. Практична цінність одержаних результатів полягає в тому, що на основі отриманих в магістерській кваліфікаційній роботі теоретичних положень запропоновано методи та розроблено програмні засоби для трасування запитів в додатках на основі архітектури мікросервісів.

Мета та завдання дослідження. Метою дослідження магістерської кваліфікаційної роботи є підвищення продуктивності додатку за рахунок додавання додаткових фільтрів і можливості додавання власних ключів в заголовок запиту додатку для розподіленого трасування запитів.

Для досягнення поставленої мети необхідно розв’язати такі завдання:

- провести аналіз предметної області, обґрунтування доцільності та постановка задачі розробки сервісу для моніторингу запитів в додатку на основі архітектури мікросервісів;

- здійснити проектування сервісу для моніторингу запитів в додатку на основі архітектури мікросервісів;

- розробити метод додавання власних ключів і значень в заголовок запиту;

- розробити метод сортування запитів з додаванням фільтрів з розширеними умовами пошуку;

- виконати програмну реалізацію сервісу для моніторингу запитів;

- провести тестування програмного продукту та виконати аналіз отриманих результатів.

Об’єктом дослідження є процес трасування запитів.

Предметом дослідження є методи та засоби відслідковування запитів в додатках на основі архітектури мікросервісів.

Методи дослідження.

Для досягнення мети дослідження застосовувалися: концепції створення сервісу для трасування запитів; концепції і методи HTTP запиту; методи створення мікро сервісів; методи об’єктно-орієнтованого програмування.

Наукова новизна одержаних результатів полягає в наступному:

1. Вперше запропоновано метод додавання власних ключів і значень в заголовок запиту при розподіленому трасуванні, особливість якого полягає у додаванні настроюваний файлів заголовку, що дає змогу масштабувати трасування запитів і, як наслідок, підвищити продуктивність додатку.
2. Вперше запропоновано метод сортування запитів, особливість якого полягає у додаванні фільтрів з розширеними умовами пошуку, що дозволяє підвищити продуктивність додатку.

Практичне значення одержаних результатів полягає в тому, що на основі проведених досліджень розроблено програмне забезпечення для трасування запитів в додатках на основі мікросервісів.

Розроблений додаток можна використовувати в якості готового продукту.

Достовірність теоретичних положень підтверджена результатами тестування розробленого додатку.

Робота складається з 5 розділів. У першому розділі проаналізовано сучасний стан питання теми розробки, проведено порівняння аналогів і обґрунтовано вибір моделі життєвого циклу для розробки ПЗ, сформульовано задачі дослідження, проаналізовано проблеми, які даних сервіс вирішує.

Другий розділ містить аналіз структури, принципів та особливостей розробки додатків, а також здійснено розробку алгоритму та структур системи.

У третьому розділі розроблено серверну і клієнтську частину, також описано програмну реалізацію додатку.

Четвертий розділ показує результати тестування розробленого додатку.

У п’ятому розділі розраховуються витрати на розробку програмного забезпечення, експлуатаційні витрати, обсяг роботи, пов'язаної з використанням програмного забезпечення, та економічний ефект від впровадження нового програмного продукту. У висновках наведені основні результати дослідження.

У додатках міститься технічне завдання, лістинг коду основних частин програми та ілюстративний матеріал до захисту магістерської кваліфікаційної роботи.

1 АНАЛІЗ ПРЕДМЕТНОЇ ГАЛУЗІ РОЗПОДІЛЕНОГО ТРАСУВАННЯ ЗАПИТІВ

1.1 Аналіз предметної галузі

Сучасні Інтернет-сервіси часто реалізуються як складні, широкомасштабні розподілені системи. Ці програми побудовані з колекцій програмних модулів які можуть бути розроблені різними командами, можливо, на різних мовах програмування і може охопити багато тисячі машин на різних фізичних об'єктах. Інструменти, які допомагають зрозуміти поведінку системи і міркування щодо питань ефективності неоцінені в такому середовищі.

Розуміння поведінки системи в цьому контексті вимагає спостереження за пов'язаними заходами у багатьох різних програмах та машинах.

Приклад пошуку в Інтернеті проілюструє деякі проблеми, з якими повинна вирішуватися така система. Служба передового доступу може поширювати веб-запит багатьом сотням сервери запитів, кожен з яких шукає в межах своєї частини індекс. Запит також може бути надісланий на інші підсистем, які можуть обробляти рекламу, перевірити правопис або шукати спеціалізовані результати, включаючи зображення, відео, новини тощо. Результати всіх цих послуг вибірково поєднуються в результатах сторінки. Загалом, тисячі машин і багато різних служб можуть необхідно обробити один універсальний пошуковий запит. Більше того, користувачі веб-пошуку чутливі до затримок, які можуть бути спричинені низькою продуктивністю в будь-якій підсистемі. Інженер, що дивиться лише на загальну затримку, може знати є проблема, але, можливо, не в змозі здогадатися, яка винна послуга, ні чому вона погано поводиться. По-перше, інженер може не знати, які саме послуги в вживанні; нові сервіси та фрагменти можуть бути додані та змінені з тижня на тиждень, обидва, щоб додати видимі для користувача функції та вдосконалити інші аспекти, такі як ефективність чи безпека. По-друге, інженер не буде експертом внутрішні органи кожної служби;

кожен будується та підтримується різною командою. По-третє, послуги та машини може одночасно ділитися багатьма різними клієнтами, тому артефакт продуктивності може бути зумовлений поведінкою ще одна програма. Наприклад, передня частина може оброблятись безліч різних типів запитів або таких систем зберігання, як Bigtable[1] може бути найбільш ефективним при спільному використанні кілька додатків[2].

Відстеження програмного забезпечення надає розробникам інформацію, корисну для налагодження. Ця інформація використовується як під час циклів розробки, так і після випуску програмного забезпечення. На відміну від реєстрації подій, відстеження програмного забезпечення зазвичай не має поняття "класу" події або "коду події". Інші причини, чому рішення для реєстрації подій на основі кодів подій не підходять для відстеження програмного забезпечення, включають:

1. Оскільки відстеження програмного забезпечення є низьким рівнем, часто існує багато інших типів повідомлень, які потрібно було б визначити, багато з яких використовуються лише в одному місці коду. Парадигма коду подій вводить значні накладні розробки для цих "єдиних" повідомлень.
2. Типи повідомлень, які реєструються, часто менш стабільні через цикл розробки, ніж для ведення журналів подій.
3. Оскільки вихідний трасування призначений для споживання розробником, повідомлення не потрібно локалізувати. Тому важливо зберігати повідомлення про кальку окремо від інших ресурсів, які потрібно локалізувати (наприклад, повідомлення про події).

Ще одне важливе значення для відстеження програмного забезпечення - це продуктивність. Оскільки відстеження програмного забезпечення є низьким рівнем, можливий обсяг повідомлень про сліди набагато вищий. Щоб вирішити проблеми з ефективністю, часто потрібно мати можливість вимкнути

відстеження програмного забезпечення або під час компіляції, або під час виконання. Інші особливі проблеми:

1. У приватному програмному забезпеченні дані трасування можуть містити конфіденційну інформацію про вихідний код продукту.
2. Якщо відстеження ввімкнено або вимкнено під час виконання, багато методів відстеження вимагають включення значної кількості додаткових даних у двійкові дані, що може побічно погіршити продуктивність, навіть якщо відстеження відключено.
3. Якщо в час компіляції ввімкнено або вимкнено відстеження, отримання даних про відстеження проблеми на машині клієнта залежить від того, чи бажає клієнт і зможе встановити спеціальну версію програмного забезпечення, що підтримує трасування, а потім дублювати проблему.
4. У багатьох напрямках трасування є дуже жорсткі вимоги до надійності. Це пояснюється як надійністю виведення слідів, так і тим, що випадки використання, які відстежуються, не повинні порушуватись.
5. В операційних системах відстеження іноді корисно в таких ситуаціях (як завантаження), коли деякі технології, що використовуються для забезпечення журналу подій, можуть бути недоступними.
6. У вбудованому програмному забезпеченні трасування вимагає спеціальних методик. [3]

Розподілене трасування швидко стає обов'язковим компонентом в інструментах, якими користуються організації для моніторингу їх складних архітектур на основі мікросервісів.

Розподілене відстеження запитів може слідкувати за запитами через кожен модуль або службу. Наприклад, розподілене трасування дозволяє архітектору додатків візуалізувати виконання кожної ітерації функції X. Розробники та фахівці з операцій, які виконують підтримку, можуть швидко визначити, що один

екземпляр функції X вводить значно більше затримок, ніж інші екземпляри, і усунути проблему.

Хоча трасування та профілювання є тісно пов'язаними дисциплінами, розподілене відстеження зазвичай розуміється як техніка, яка використовується для зв’язку інформації про різні одиниці роботи разом - зазвичай виконуються в різних процесах або хостах - щоб зрозуміти цілий ланцюжок подій. У сучасній програмі це означає, що розподілене трасування може використовуватися для розповіді про історію HTTP-запиту, коли він проходить через безліч мікросервісів.

Поширене трасування залежить від слідів та прольотів. Слід - це повна обробка запиту. Трасування представляє всю подорож запиту під час руху через усі сервіси або компоненти розподіленої системи. Усі події відстеження, породжені запитом, мають ідентифікатор сліду, який інструменти використовують для організації, фільтрації та пошуку конкретних слідів.

Кожен слід складається з декількох прольотів. Проміжок, який іноді називають сегментом, - це діяльність або операція, яка відбувається в межах окремих служб або компонентів розподіленої системи. Кожен проміжок часу - ще один крок у загальній обробці загального запиту. Проміжки, як правило, називаються і приурочені до операцій. Кожен проміжок містить унікальний ідентифікаційний проміжок, а також може містити метадані або інші примітки.

1.2 Вибір моделі життєвого циклу для розробки додатку

При розробці будь-якого програмного додатку, необхідним етапом є вибір моделі життєвого циклу.

Модель життєвого циклу – це структура, що складається з процесів, робіт та задач, які включають в себе розробку, експлуатацію та супровід програмного продукту і охоплює життя системи від визначення вимог до неї до припинення її використання.

На сьогодні, найбільшого розповсюдження набули три моделі:

1 Каскадна (водоспадна).

2 Ітеративна.

3 Спіральна (модель Боема).

Каскадна або водоспадна модель припускає строго послідовне у часі та одноразове виконання всіх фраз проекту з жорстким попереднім плануванням в контексті наперед визначених вимог до програмної системи(рис. 1.1).

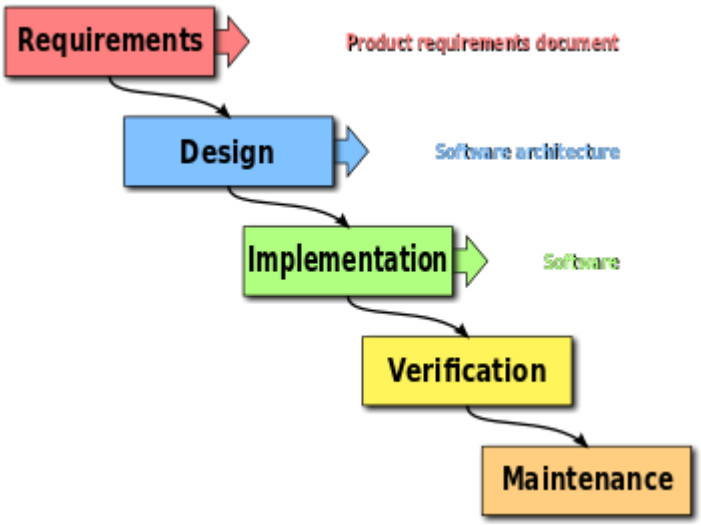


Рисунок 1.1 – Каскадна модель

Основними перевагами каскадної моделі є: стабільність вимог протягом усього життєвого циклу розробки; можливість послідовного усунення виникаючих складнощів; визначеність та зрозумілість кроків моделі та простота її застосування; спрощення можливості здійснення планування, контролю та управління проектом; ефективність для проектів з чіткими і зрозумілими, але важко реалізованими вимогами; ефективність для проектів з високими вимогами до якості при відсутності жорстких обмежень витрат і графіка робіт.

Ітеративна модель припускає розбивку життєвого циклу на послідовність ітерацій, кожна з яких нагадує «міні-проект», включаючи всі фази життєвого циклу в застосування для створення менших фрагментів функціоналу. Основна ідея полягає в тому, щоб розробити систему шляхом циклів, що повторюються, даючи змогу розробнику скористатися перевагами того, що було вивчене під час розробки попередніх частин або версій системи(рис. 1.2).

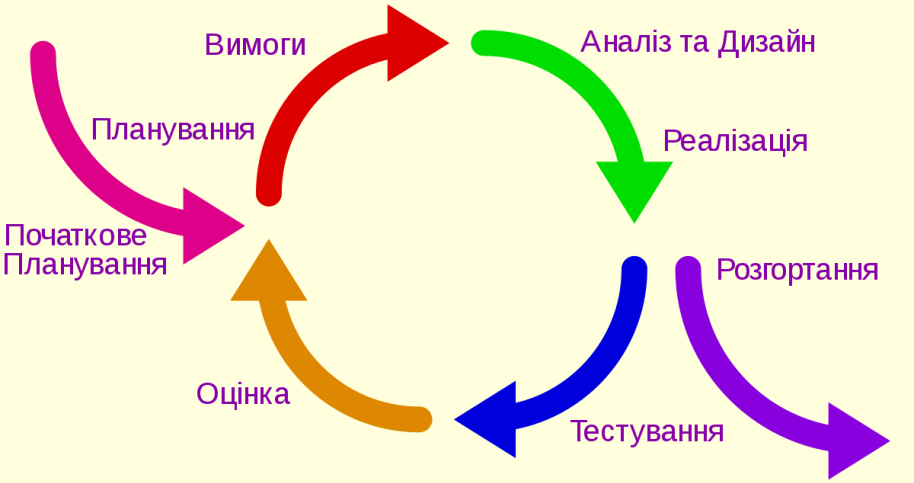


Рисунок 1.2 – Ітеративна модель

Основними перевагами ітеративної моделі є: зниження ризиків у результаті виявлення конфліктів між вимогами, моделями та реалізацією проекту та управління їх вирішенням; організація ефективного зворотнього зв'язку проектної команди зі споживачем; швидкий випуск мінімального цінного продукту і можливість вивести продукт на ринок і почати експлуатацію набагато раніше.

Спіральна модель – це модель, відмінною особливістю якої є спеціальна увага ризикам, що впливає на організацію життєвого циклу. Ця модель характерна при розробці новаторських систем(рис. 1.3). На початку роботи над проектом у замовника та розробника немає чіткого бачення підсумкового продукту або стовідсоткової впевненості в успішній реалізації проекту. В зв'язку з цим

приймається рішення розробки системи по частинах з можливістю зміни вимог або відмови від подальшого розвитку [4].

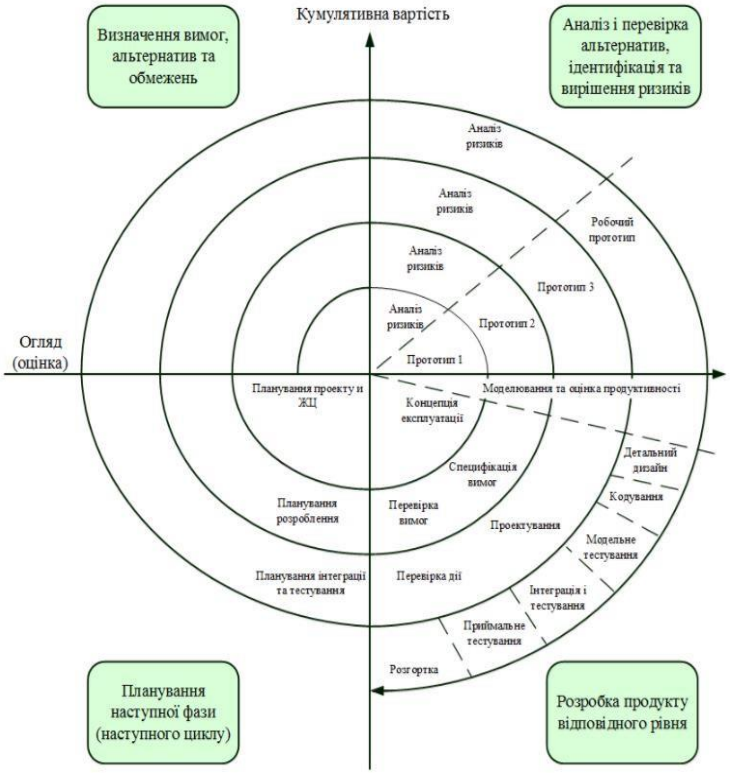


Рисунок 1.3 – Спіральна модель

Основними перевагами спіральної моделі є те, що вона: дозволяє швидше показати користувачам системи працездатний продукт і тим самим активізувати процес уточнення та доповнення вимог; допускає зміну вимог при розробці інформаційної системи; забезпечує гнучкість в управлінні проектом; дозволяє отримати більш надійну та стійку систему; дозволяє удосконалювати процес розробки та зменшує ризики замовника.

Проаналізувавши популярні моделі життєвого циклу програмного забезпечення, було вирішено використовувати каскадну модель для розробки програмного продукту. Цю модель найчастіше використовують при розробці невеликих продуктів і за наявності чітких вимог та термінів виконання, оскільки вона є простою в застосуванні і дозволяє підтримувати стабільність.

1.3 Характеристика і аналіз аналогів

У процесі літературного огляду програмного забезпечення, яке функціонує в області прогнозування трасування запитів, доцільно зробити ряд уточнень:

Більшість інструментів, перелічених тут, можна класифікувати як бібліотеку приладів, трекер, інструмент аналізу (бекенд + інтерфейс користувача) або будь-яку їх комбінацію.

Майже всі реалізації побудовані на основі специфікації OpenTracing.

Оскільки OpenTracing повинен працювати на багатьох мовах, цей стандарт описує взаємодію з трасуванням запитів через інтерфейс.

Одним з розглянутих аналогів є Apache (Incubating) Zipkin - Zipkin був спочатку розроблений у Twitter та відкритий джерелом у 2012 році. Це одна з найзріліших систем трасування з відкритим кодом та надихнула майже всі сучасні інструменти розповсюдження трасування. Це повне рішення для відстеження, включаючи бібліотеки приладів, трекер та інструмент аналізу. Формат розповсюдження, який називається B3, - це поточна лінгва-франка в розподіленому трасування, а також його формат даних, який в основному підтримується іншими інструментами, такими як Envoy Proxy на стороні виробника та інші рішення трасування на споживчій стороні. Однією з сильних сторін Zipkin є кількість високоякісних рамкових бібліотек інструментарію.

Ще одним рішенням в трусування запитів є Jaeger.

Спочатку Jaeger був розроблений в Uber, відкритий в 2017 році, а незабаром перейшов до Cloud Cloud Native Computing Foundation (CNCF). Натхнення Dapper та Zipkin можна побачити в оригінальній архітектурі, моделі даних та номенклатурі Jaeger , але вона розвинулася поза цим. Для інструментальної частини Jaeger використовує API OpenTracing, який з самого початку був інструментарієм першого класу. Інструмент аналізу дуже легкий, що робить його ідеальним для цілей розвитку та для високо еластичних середовищ (наприклад, багатонаціональних кластерів Kubernetes), і він є інструментом пошуку за замовчуванням для таких інструментів, як Istio.

1.4 Проблеми які можна вирішити з підтримкою трасування

1. Знайти вузькі місця у виробництві, як всередині однієї служби, так і у всіх деревах виконані між усіма учасниками обслуговування. Наприклад:

- Багато коротких послідовних визовів між сервісами, наприклад, на геокодинах або на базах даних.
- Довгі очікування введення виводу, наприклад, передані дані по мережах або чтеніе з дискою.
- Довгий парсинг даних.
- Довгі операції, вимагаючі процесора.
- Учасники кодексу, які не потрібні для напівпроцедур кінцевого результату і можуть бути видаленими, або запущені.

2. Наглядно зрозуміти в якій послідовності, що виходить і що відбувається, коли виконується операція(рис 1.4).

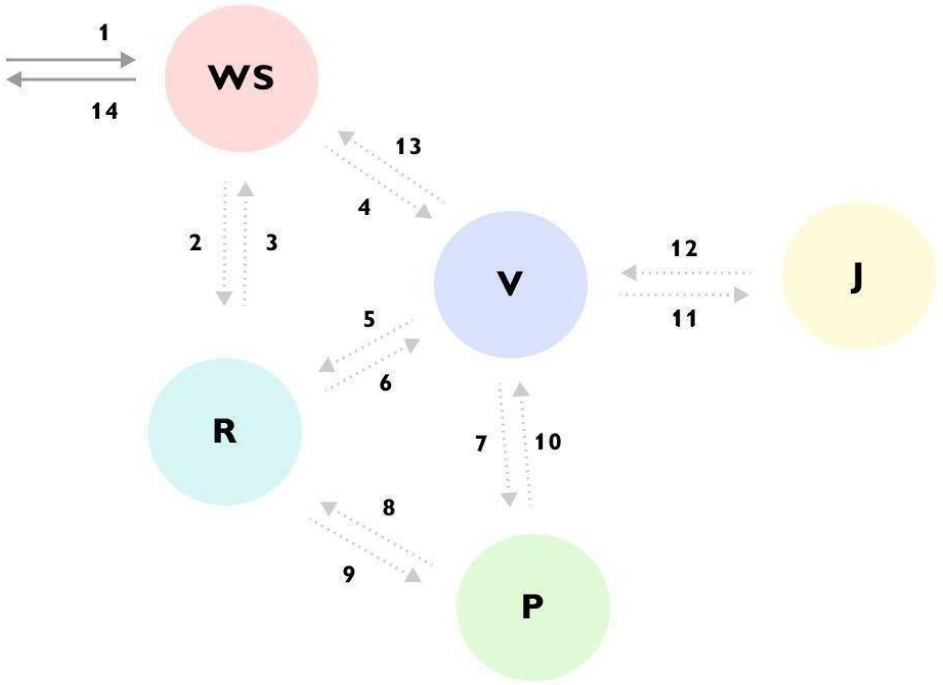


Рисунок 1.4 – Загальна схема процесу трасування

Наприклад, запит прийшов у службу WS -> сервіс WS доповнив дані через сервіс R -> далі виправити запит у сервіс V -> сервіс V завантажив багато даних з сервісу R -> перейшов у службу P -> сервіс P ще раз зайшов в сервісі R -> сервіс V прогнозував результат і поширився в сервісі J -> і тільки в такому випадку попередив відповідь у сервісі WS, при цьому продовжившись у фоновому режимі, обчислювати щось ще.

Без такого Трейса або докладної документації на весь процес дуже складно зрозуміти, що відбувається, перший раз поглянувши на код, та й код розкиданий по різних сервісів і прихований за купою бінов і інтерфейсів.

3. Збір інформації про дерево виконання для подальшого відкладеного аналізу. На кожному етапі виконання в трейс можна додати інформацію, яка доступна на даному етапі і далі розібратися які вхідні дані привели до подібного сценарію. наприклад:

- ID користувача
- права
- Тип обраного методу
- Лог або помилка виконання

4. Перетворення трейсів в підмножину метрик і подальший аналіз вже в вигляді метрик.

1.5 Постановка задачі

Розглянуті рішення мають закриті алгоритми, що не дозволяє вивчити методи, які використовуються в них для аналізу даних і прогнозування. Це обумовлює складність їх впровадження в уже існуючу систему в якості окремих аналітичних модулів.

У зв'язку з цим необхідно розробити програмне забезпечення для візуального відображення проходу запиту.

1.6 Висновок

Таким чином, в результаті аналізу предметної області та засобів розробки програмних засобів для розподіленого трасування можна зробити наступні висновки:

- функціонування програмних засобів трасування запитів дозволить знайти вузькі місця у виробництві, як всередині однієї служби, так і у всіх деревах виконані між усіма учасниками обслуговування;
- для програмної розробки програмного продукту доцільно використовувати системи управління реляційними базами даних, що дозволить забезпечити збереження достовірної інформації, а також програмний додаток для роботи користувача;
- для розробки програмного продукту сформовано технічне завдання, яке враховує всі вимоги до певних областей функціонування програмного забезпечення прогнозування трасування запитів.

2 РОЗРОБКА МОДЕЛЕЙ І МЕТОДІВ ДЛЯ МОНІТОРИНГУ ЗАПИТІВ

2.1 Розробка серверної частини

Розробка структурної архітектури програми є одним з найбільш важливих етапів у процесі розробки програмного забезпечення, неправильний вибір архітектури веде до ризику зриву всього проекту в майбутньому. Даний етап є базовим для всього процесу розробки. Продумана архітектура дозволяє легко модифікувати програмний продукт, якщо відбудуться зміни вимог до нього.

Архітектура – це набір значущих рішень з приводу організації системи програмного забезпечення, набір структурних елементів та їх інтерфейсів, за допомогою яких компонується система, разом з їх поведінкою, обумовленим у взаємодії між цими елементами, компонування елементів в поступово 64 укрупнюючих підсистеми, а також стиль архітектури який направляє цієї організації – елементи та їх інтерфейси, взаємодії і компонування [5]. Під архітектурою розуміється сукупність компонентів програми, а також зв'язку і способи організації інформаційного обміну між ними. Логічна структура описує об'єкти, які пов'язані з іншими об'єктами. Вона визначає зв'язки між ними.

Мабуть, простіше почати з того, що серверна частина - це не завантаження чи програма. Розподілене трасування вимагає, щоб розробники програмного забезпечення додавали інструментарій до коду програми або до фреймворків, які використовуються в додатку.

Як розподілене трасування, зазвичай використовується в архітектурі мікросервісів. Мікросервіси — архітектурний стиль за яким єдиний застосунок будується як сукупність невеликих сервісів кожен з яких працює у своєму власному процесі і комунікує з рештою використовуючи легковагові механізми, зазвичай HTTP. Ці сервіси будуються навколо бізнес-потреб і розгортаються незалежно з використанням зазвичай повністю автоматизованого середовища.

Існує абсолютний мінімум централізованого керування цими сервісами. Самі по собі вони можуть бути написані з використанням різних мов і технологій зберігання даних.[6]

Як виглядають дані відстеження?

Починаючи інструментарій, корисно скласти уявну картину форми даних, які ви надсилатимете. То що йде в типовий слід? Слід складається з багатьох прольотів (їх також називають подіями). Кожен проміжок представляє одну одиницю роботи для послуги, програми чи фрагмента коду. Наприклад, один запит до бази даних або один запит HTTP до сторонньої служби може бути представлений як проміжок. Проміжки можуть мати стосунки батько-дитина один з одним, тому ви можете побачити слід, що представляє HTTP-запит до сервера додатків; цей слід може включати дочірній проміжок, що представляє HTTP-запит до внутрішньої служби, і тоді в цьому дочірньому діапазоні може бути багато власних дочірніх прольотів, що представляють різні запити бази даних, які він запускав.

Ось деякі типові поля, які ви побачите, які допомагають описати ці проміжки:

Span ID: довільний унікальний ідентифікатор, який використовується для ідентифікації цієї конкретної одиниці роботи;

Trace ID: довільний унікальний ідентифікатор, що описує слід, до якого належить цей проміжок. У типовій додатковій програмі трасування може представляти один HTTP-запит користувача до веб-програми. У веб-переглядачі є більша ймовірність представляти собою більшу одиницю роботи, як-от перегляд сторінки;

Parent span ID (ідентифікатор батьківського прольоту): це поле використовується для малювання причинно-наслідкового зв'язку між двома проміжками. Наприклад, якщо ваша веб-програма надсилає http-запит до бекенд-сервісу, який потім запускає запит до бази даних, інтервал для запиту бази даних

буде використовувати ідентифікатор span інтервалу служби резервного сервера як його ідентифікатор батьківського інтервалу;

Name (назва): конкретний тип робіт, який цей проміжок представляє, напр. "Render json", "select", "save to S3" тощо;

service name (назва служби): область бази коду, з якої походить цей проміжок. У веб-переглядачі це може допомогти вам визначити певний пакет або область вашого коду ("маршрутизатор", "скорочення", "асинхронний рендер").

duration (тривалість): кількість часу, яку ця конкретна одиниця роботи зайняла, в мілісекундах.

Основна ідея або мотив MSA (Microservices Architecture) - це розподіл або розбиття будь-якого великого та / або складного монолітного програмного забезпечення на кілька менших автономних модулів (згаданих як Microservices), де кожен мікросервіс (тобто автономний модуль) можна проектувати, будувати та перевіряти індивідуально. Після того, як всі мікросервіси будуть готові (тобто побудовані індивідуально найкраще, наскільки кожен може бути перевірений і перевірений окремо), розгортайте додаток, використовуючи механізм, такий як SOA (Service Oriented Architecture) для вільного з'єднання всіх мікропослуг.

Можна виділити основні властивості цієї архітектури:

1. Високий рівень незалежності: незалежна розробка, незалежне розгортання
2. Незалежне масштабування
3. Невелика кодова база зменшує кількість конфліктів та дозволяє швидко залучувати нових розробників
4. Простота заміни однієї реалізації сервісу іншою
5. Простота додавання нового функціоналу в систему
6. Ефективне використання ресурсів
7. Еластичність: вихід з ладу одного сервісу зазвичай не призводить до виходу з ладу всієї системи
8. Сервіси організовані відносно бізнес логіки яку вони виконують

9. Кожен сервіс незалежно від інших може бути реалізований за допомогою будь-якої мови програмування, СБД, та ін.
10. Архітектурно побудовані за симетричним принципом (виробник-споживач)

З точки зору розподіленої системи трасування на рівні додатків, програмна система має вигляд наступної схеми(рис 2.1).

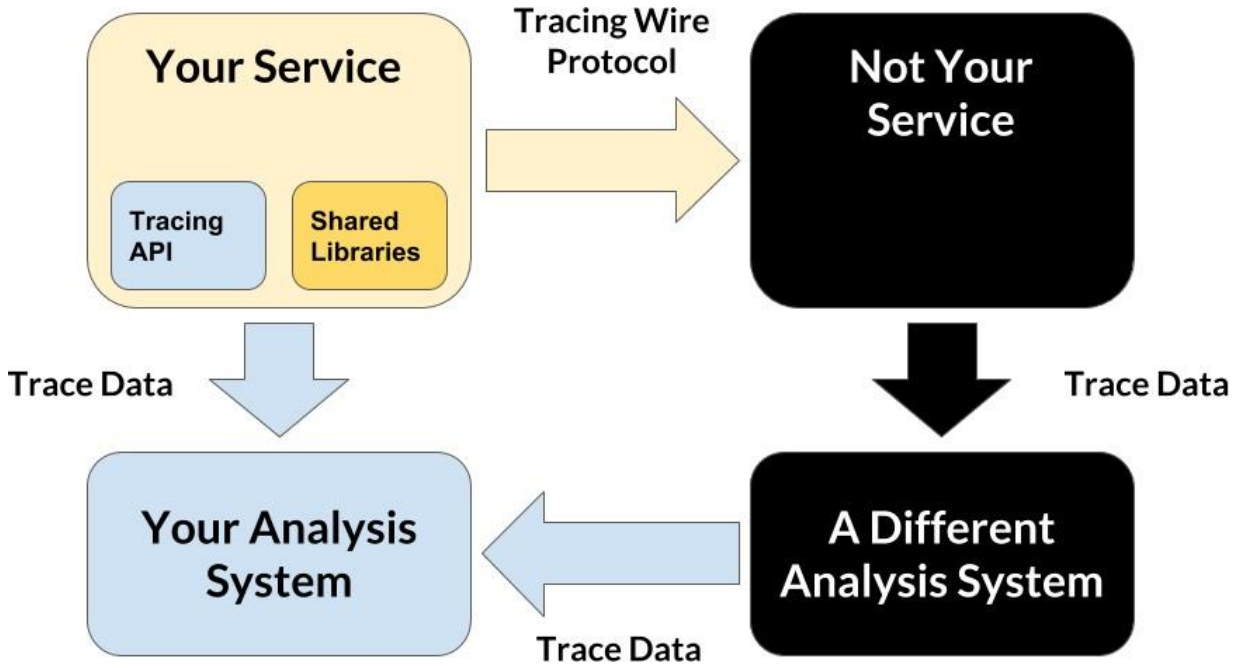


Рисунок 2.1 – Загальна програмна система процесу трасування

Підключення в сервісі зводиться до того, щоб створити інтерфейс io.opentracing.Tracer, після чого все трейсі через нього будуть відправлятись в реальний агент(рис. 2.2).

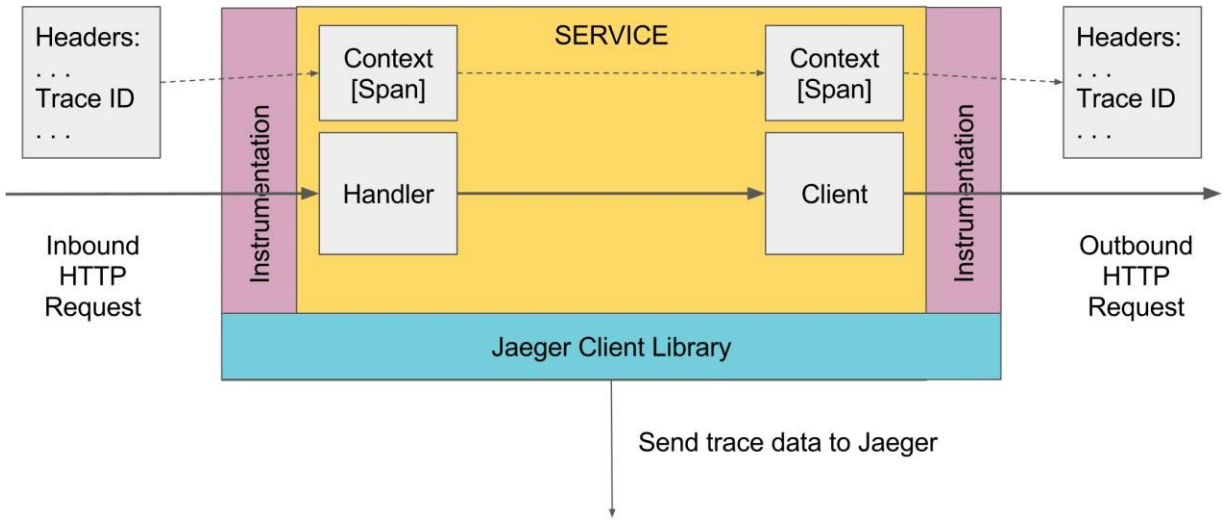


Рисунок 2.2 – Встановлення клієнту в сервісі

2.2 Розробка візуальної клієнтської частини

Майже завжди при написанні клієнтської частини, ми говоримо про реалізацію MVC архітектури, але деякі його вважають патерном.

MVC пропонує архітектурні переваги порівняно зі стандартним JavaScript - це допомагає писати краще організований і, отже, більш доступний код. Ця закономірність була використана і широко перевірена на декількох мовах і поколіннях програмістів.

MVC складається з трьох компонентів(рис. 2.3):

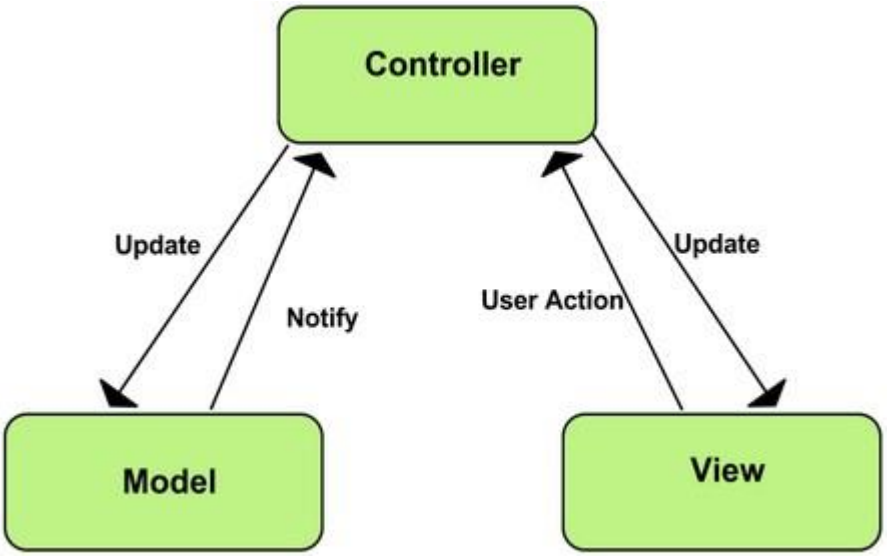


Рисунок 2.3 – Компоненти MVC

Модель.

Модель — це місце, де зберігаються об’єкти даних програми. Модель нічого не знає про види та контролери. Коли модель змінюється, зазвичай вона повідомляє своїх спостерігачів про те, що зміни відбулися.

Щоб зрозуміти це далі, скористайтеся додатком списку To-do, простим веб-додатком на одній сторінці, який відстежує ваш список завдань(рис. 2.4).

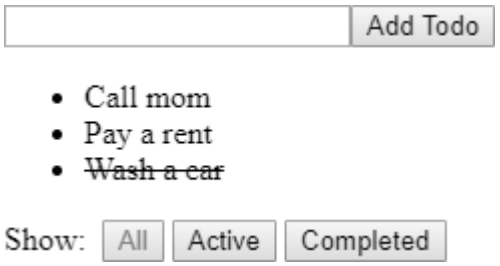


Рисунок 2.4 – To-do лист

Тут модель представляє атрибути, пов'язані з кожним елементом To-do, такі як опис та статус. Коли створюється новий елемент To-do, він зберігається в екземплярі моделі.

Вид.

Це те, що представлено користувачам і як користувачі взаємодіють із додатком. Перегляд робиться за допомогою HTML, CSS, JavaScript та часто шаблонів. Ця частина додатка Chrome має доступ до DOM.

Наприклад, у вищевказаному веб-додатку для списку To-do ви можливо створити представлення, яке добре відображає список елементів To-do користувачам. Користувачі також можуть ввести новий елемент To-do через деякий формат введення; проте представлення не знає, як оновити модель, оскільки це завдання контролера.

Контролер.

Контролер є органом, що приймає рішення, і клеєм між моделлю та видом. Контролер оновлює подання, коли модель змінюється. Він також додає слухачів подій до перегляду та оновлює модель, коли користувач маніпулює поданням.

У веб-програмі списку To-do, коли користувач перевіряє елемент як виконаний, клацання пересилається до контролера. Контролер змінює модель для позначення елемента як закінченого. Якщо дані потрібно зберігати, вони також зберігають асинхронізацію на сервері. У розширеній розробці веб-додатків на стороні клієнта, таких як Chrome Apps, також важливим є збереження даних, збережених у локальному сховищі. У цьому випадку контролер також обробляє збереження даних у сховищі на стороні клієнта, наприклад, FileSystem API.

Існує кілька варіантів структури дизайну MVC, таких як MVP (Model – View – Presenter) та MVVP (Model – View – ViewModel). Навіть із так званою схемою дизайну MVC існує певна різниця між традиційною схемою MVC та сучасною інтерпретацією у різних мовах програмування. Наприклад, деякі рамки, засновані на MVC, матимуть вигляд спостерігати за змінами в моделях, а інші дозволяють контролеру обробляти оновлення перегляду[7][8].

Підсумовуючи, модель MVC надає модульність розробникам додатків, і це надає[8]:

1. Код для багаторазового використання та розширення
2. Відокремлення логіки зору від логіки бізнесу.
3. Дозвольте одночасну роботу між розробниками, які відповідають за різні компоненти (наприклад, рівень інтерфейсу та логіка ядра).
4. Простіше в обслуговуванні.

Для написання веб-додатку для розподіленого трусавання запитів фрейморки React та Redux.

React — відкрита JavaScript бібліотека для створення інтерфейсів користувача, яка покликана вирішувати проблеми часткового оновлення вмісту веб-сторінки, з якими стикаються в розробці односторінкових застосунків. Розробляється Facebook, Instagram і спільнотою індивідуальних розробників[9].

React дозволяє розробникам створювати великі веб-застосунки, які використовують дані, котрі змінюються з часом, без перезавантаження сторінки. Його мета полягає в тому, щоб бути швидким, простим, масштабованим. React обробляє тільки користувацький інтерфейс у застосунках. Це відповідає видові у шаблоні модель-вид-контролер (MVC), і може бути використане у поєднанні з іншими JavaScript бібліотеками або в великих фреймворках MVC, таких як AngularJS. Він також може бути використаний з React на основі надбудов, щоб піклуватися про частини без користувацького інтерфейсу побудови веб-застосунків. Як бібліотека інтерфейсу користувача React часто використовується разом з іншими бібліотеками, такими як Redux.

Однією з основних переваг React над іншими фреймворками і бібліотеками є використання віртуального дому(Virtual DOM). Віртуальний DOM (VDOM) - це концепція програмування, в якій ідеальне або "віртуальне" подання інтерфейсу зберігається в пам'яті та синхронізується з "реальним" DOM бібліотекою, такою як ReactDOM. Цей процес називається примиренням.

Цей підхід дозволяє декларативному API реагувати: ви повідомляєте

React, який стан ви хочете, щоб користувач інтерфейсу був, і він гарантує, що DOM відповідає цьому стану. Це абстрагує маніпуляції атрибутами, обробку подій і ручне оновлення DOM, які в іншому випадку доведеться використовувати в додатку(рис 2.5).

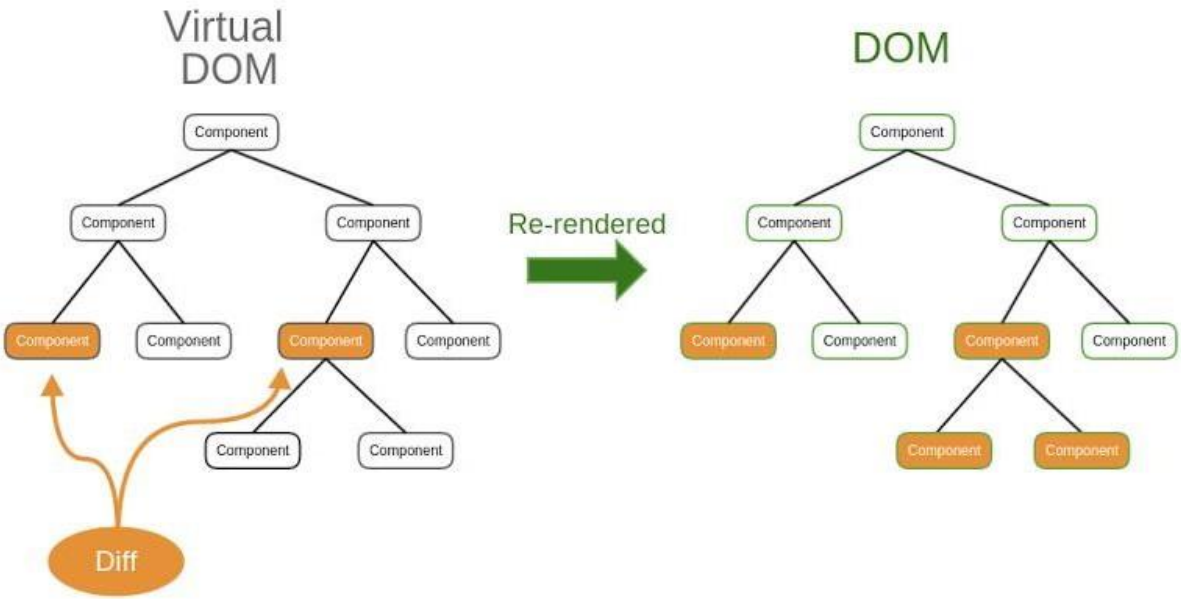


Рисунок 2.5 – Приклад роботи віртуального дому

Redux — бібліотека відкритих вихідних кодів JavaScript для управління станом програми. Це найчастіше використовується з бібліотеками, такими як React або Angular, для побудови користувацьких інтерфейсів. Подібний (і натхненний) архітектурою Facebook Flux, він був створений Деном Абрамовичем та Ендрю Кларком.

З появою React 16, було введено Fiber. Це новий двигун примирення в React 16. Його основна мета полягає в тому, щоб активізувати поступову візуалізацію віртуальної DOM.

Fibers - набагато нижчий рівень абстракції, ніж зазвичай думають розробники додатків. Головна мета Fiber - надати можливість React скористатися перевагами планування. Зокрема:

1. Призупинити роботу і повернутися до неї пізніше.

2. Присвоїти пріоритет різним видам робіт.
3. Повторно використовувати раніше закінчені роботи.
4. Перервати роботу, якщо вона більше не потрібна.

Для того, щоб зробити що-небудь із цього, спочатку потрібен спосіб розбити роботу на одиниці. В одному сенсі, ось що таке волокно. А волокно являє собою одиницю роботи.

Архітектура Fiber названа на честь алгоритму, що лежить в її основі. Алгоритм полягає у розбитті процесу оновлення на дві фази:

Фаза узгодження (reconciliation) — коли виконуються переобрахунки компонентів і відбувається оновлення DOM у пам’яті.

Фаза внеску (commit) — коли виконується безпосереднє оновлення DOM.

Варто зазначити, що фазу узгодження (reconciliation) можна переривати. fiber за допомогою requestIdleCallback просить у браузера виділити час, коли той не буде завантажений роботою. При зворотньому виклику браузер вказує, скільки, власне, в нього є вільного часу. Це дає змогу fiber-у планувати частину оновлень на цей проміжок. Якщо браузер не підтримує requestIdleCallback, то React робить поліфіл (polyfill).

Алгоритм Fiber у свою чергу названий на честь найменшого об’єкта, що лежить в його основі. За кожним екземпляром (компонента чи елемента) закріплений такий об’єкт, що контролює його стан та зв’язок з іншими компонентами[10].

Redux - невелика бібліотека з простим, обмеженим API, розробленим як передбачуваний контейнер для стану програми. Він діє подібним чином на функцію зменшення, концепцію функціонального програмування. На це впливає функціональна мова програмування Elm [11].

Redux використоується в розроблюваному додатку для зберігання стану всього додатку тому як зберігати стан для декількох компонентів в React дуже

незручно(рис 2.6)

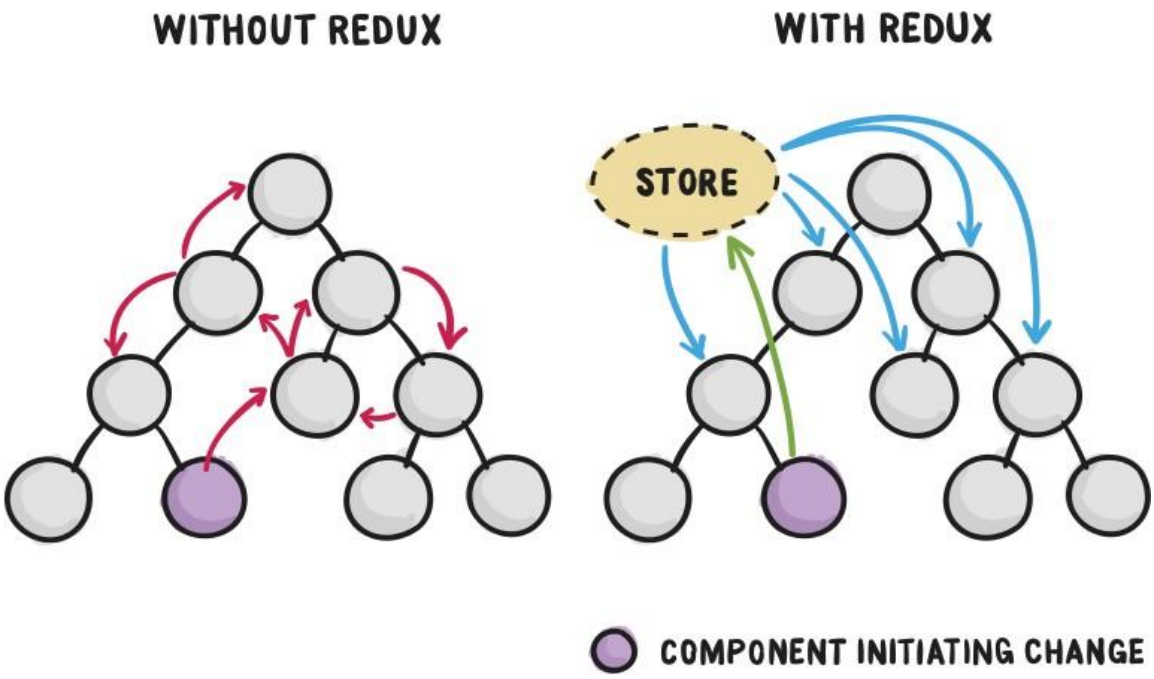


Рисунок 2.6 – Використання Redux з React компонентами

Згідно з офіційною документацією, Redux - це передбачуваний контейнер для додатків JavaScript. Отже, це в основному три особливості:

1. Чиста функція
2. Глибина агностичнійсть підписки на сховище
3. Простота тестування

Чиста функція

Визначення: Функція, яка працює виключно на переданих в неї аргументах і не має побічних ефектів, - це чиста функція(рис. 2.7).

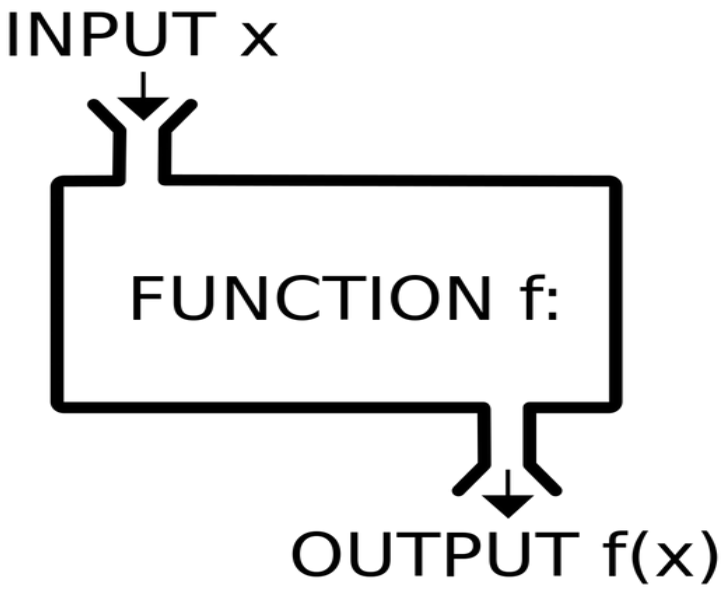


Рисунок 2.7 – Визначення чистої функції

У математиці ми маємо $sin\ (x)$. У JavaScript ми маємо $sum = (a, b) => a + b.$

Оскільки функція sum працює виключно з вхідними даними, задані аргументи однакові, то вони дадуть один і той самий результат, незалежно від кількості разів або за яких умов ми його викликаємо.

Як і в, $sum(4,\ 5)$ завжди повернеться 9. Знову і знову. Без змін або модифікації аргументів. Так і *sin* $sin\ (x)$ наша сума (x, y) є передбачуваною.

Глибина агностичнійсть підписки.

Скажімо, у вас є функція пошуку, яка надає SearchBar, який в кінцевому рахунку надає набір компонент SearchItems. I <SearchBar />, і <SearchItems /> можуть безпосередньо підписатися на сховище(рис. 2.8).

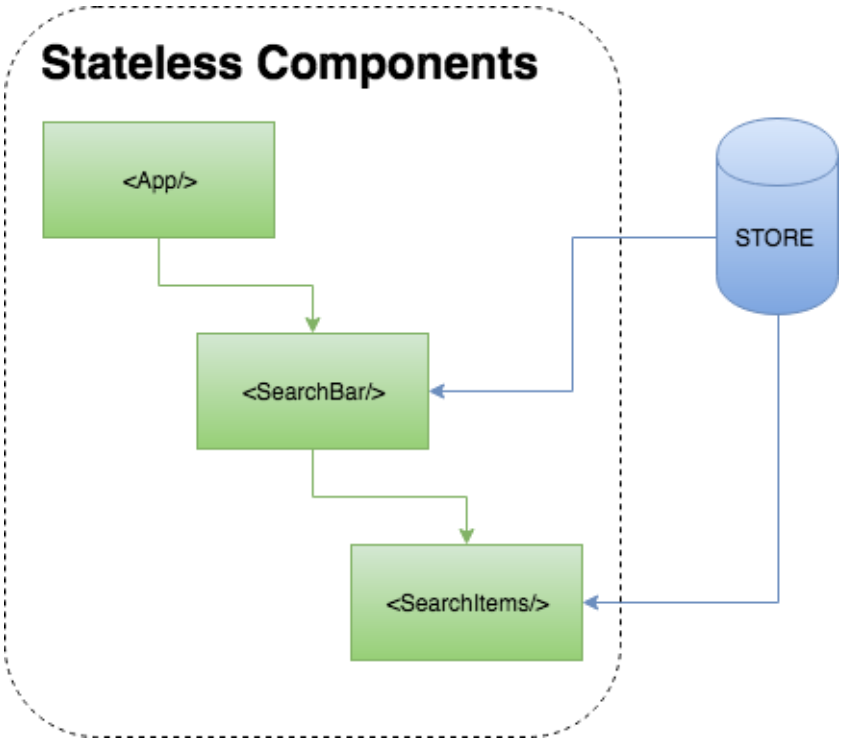


Рисунок 2.8 – Глибина агностичнійсть підписки

Простота тестування.

Оскільки редуктори - це всі чисті функції, ми не вимагаємо ніяких зухвалих зусиль, коли ми пишемо для них одиничні тести.

Просто передайте в дію та очікуйте моментальний знімок стану замість нього. Тепер ваша логіка додатків може бути повністю компонентно тестованою.

2.3 Метод додавання власних ключів та значень в заголовок запиту при розподіленому трасуванні

Заголовки HTTP дозволяють клієнту та серверу передавати додаткову інформацію із запитом або відповіддю HTTP. Заголовок HTTP складається з його не чутливий до регістру імені, за яким слідує двокрапка (:), потім значення. Пробіл перед значенням ігнорується.

Власні заголовки раніше використовувались з префіксом X, але ця конвенція була застаріла в червні 2012 року через незручності, які вона викликала, коли нестандартні поля стали стандартними в RFC 6648; інші

перераховані в реєстрі IANA, оригінальний вміст якого визначено в RFC 4229. IANA також підтримує реєстр запропонованих нових заголовків HTTP.

Заголовки можна згрупувати відповідно до їх контексту:

1. Загальні заголовки застосовуються як до запитів, так і до відповідей, але не мають відношення до даних, що передаються в тілі.
2. Заголовки запитів містять більше інформації про ресурс, який потрібно отримати, або про клієнта, який запитує ресурс.
3. Заголовки відповідей містять додаткову інформацію про відповідь, як-от її місцезнаходження або про сервер, який її надає.
4. Заголовки об'єктів містять інформацію про тіло ресурсу, як його довжина вмісту або тип MIME.[12]

Як було описано вище, коли сервіс A отримує запит, який не містить інформації про відстеження, додаток для відстеження запустить новий слід, призначить йому випадковий ідентифікатор сліду та прийме рішення вибірки на основі поточно встановленої стратегії вибірки. Рішення вибірки поширюватиметься із запитами до B та C, тож ці служби не прийматимуть рішення про вибірку знову, а натомість будуть дотримуватися рішення, прийнятого вищою службою A.

Для масштабування трасування додана можливість додавати ключі при проходженні запиту через клієнт. Це дає можливість масштабувати трасування. Ключі та значення будуть зберігайся в форматі JSON.

JavaScript Object Notation (JSON) це формат обміну даними. Незважаючи на те, що JSON не є строгим підмножиною, він нагадує синтаксис JavaScript. Хоча багато мов програмування підтримують JSON, він особливо корисний для JavaScript-додатків, таких як веб-сайти і розширення для браузера. JSON може представляти числа, булеві значення, рядки, null, масиви (впорядковані послідовності значень) і об'єкти (складаються з пар ключ-значення). Ключем є рядок, значенням може виступати будь-який примітив, масив або об'єкт. JSON

спочатку не надає підтримки більш складних типів даних, таких як функції, регулярні вирази, дати і так далі. Об'єкти дат за замовчуванням серіалізіруються в рядки, що містять дату в ISO-форматі, тому дані не будуть загублені.

Для додання ключів і значень необхідно:

1. При встановленні клієнту трасування, створити файл у форматі JSON з самими ключами та значеннями.
2. Підключити заданий файл у опції при ініціалізації клієнту:

```
import initTracer from 'scr/tracer'

import customHeaders from './customHeaders.json'

var config = {
  serviceName: 'my-diploma-project',
};
var options = {
  tags: {
    'my-diploma-project': '1.0.0',
  },
  // Adding custom headers.
  customHeaders: customHeaders,
  metrics: metrics,
  logger: logger,
};
var tracer = initTracer(config, options);
```

3. Далі під час роботи клієнту, наприклад, можливо змінювати ключі і значення для відлагодженя сервісу.

2.4 Метод сортування запитів із додаванні фільтрів з розширеними умовами пошуку

Додавання фільтрів в візуальній частині додатку буде здійснюватись за допомогою сторонніх бібліотек, що значно спростить розробку та підвищить швидкість, надійність додатку. Завантаження цих бібліотек буде здійснюватися з npm репозиторіїв.

npm - це найбільший у світі реєстр програмного забезпечення. Розробники з відкритим кодом з усіх континентів використовують npm для обміну та

запозичення пакетів, а багато організацій використовують npm і для управління приватною розробкою.

npm складається з трьох різних компонентів:

1. веб-сайт
2. інтерфейс командного рядка (CLI)
3. реєстр

Веб-сайт використовується, щоб відкрити пакети, налаштувати профілі та керувати іншими аспектами npm-досвіду. Наприклад, можна створити організації для управління доступом до публічних або приватних пакетів.

CLI працює від терміналу, і це, як більшість розробників взаємодіє з npm.

Реєстр - це велика загальнодоступна база даних програмного забезпечення JavaScript та метаінформації, що його оточує.

Для додавання фільтрів було використано бібліотеку - react-select. Було додано два нових фільтри:

1. Тривалості запит.
2. На час виконання запиту.

Фільтр для власних ключів, що були описані вище, який дає змогу пошуку запитів по встановленим заздалегідь значенням та ключам, це дозволить ефективно використовувати додання власних ключів.

Фільтр тривалості запиту, який надає можливість встановити інтервал часу, мінімальний час або максимальний час, що дозволить легко відслідковувати запити, які задовольняють цю умову.

Фільтр часу виконання запиту, надає можливість знайти ті запити які відбулися в певний момент часу, наприклад, дві години назад.

Для роботи з часом було використано бібліотеку moment.js. moment.js був розроблений для роботи як у браузері, так і в Node.js.

Весь код повинен працювати в обох цих середовищах, і всі тести блоку виконуються в обох цих середовищах.

Дату було вирішено зберігати у форматі ISO 8601[13]. Дата має наступний формат - 2019-12-08T09:59:18+00:00.

Отже, алгоритм роботи фільтрів такий:

1. Після встановлення фільтру часу або фільтр тривалості запиту, обране значення зберігається в сховищі Redux.
2. Якщо одне з значень змінено, то виконується подія для оновлення цих значень, відповідно.
3. При натисненні кнопки “Find”, збереженні данні вибираються з сховища та будуються у RSQL. Після чого виконується запит до серверної частини разом з збудованими параметрами.
4. На серверній частині відбувається роздріб переданий параметрів, після чого відбувається побудування запиту до бази даних для вибору запитів, які задовольняють умови пошуку.
5. Після вибору запитів будується відповідь у форматі JSON, та відправляється назад.

2.5 Висновок

В даному розділі було описано спосіб побудови сервісу для моніторингу запитів в додатку на основі архітектури мікро сервісів, визначено технології для побудови клієнтської та серверної частини. Також було описано дві науконі новизни: метод сортування запитів із додаванні фільтрів з розширеними умовами пошуку; Метод додавання власних ключів та значень в заголовок запиту при розподіленому трасуванні.

3 ПРОГРАМНА РЕАЛІЗАЦІЯ ДОДАТКУ ДЛЯ МОНІТОРИНГУ ЗАПИТІВ

3.1 Обґрунтування вибору мови програмування

Додаток розробляється в середовищі TypeScript. Для відповіді обґрунтування вибору було проведено порівняння TypeScript з іншими мовами програмування Web-додатків. Це його основні конкуренти - Perl, ASP.NET, ColdFusion, Java і JS.

Оскільки TS є надстройкою над JS, далі буде проводитися порівняння JavaScript з наведенеми анагогами вище, а вже в останню чергу TS і JS.

1. JS і Perl

Мова Perl був створений програмістом Ларрі Уоллом (Larry Wall) і розшифровується як Practical Extraction and Report Language. Перевести це можна як "мова для практичного вилучення даних та складання звітів". Perl є інтерпретатором і з'явився набагато раніше, ніж Web. З появою інтернету мову Perl виявився зручним інструментом для створення динамічних Web-сторінок. Великого поширення Perl отримав тому, що був доступний на кожному Web-сервер, оскільки практично всі вони працювали під управлінням UNIX, і альтернативою йому був лише більш складну мову C.

Perl програє JS в витонченість, притому, що JS пропонує функціональність, аналогічну Perl при значно меншій надлишковості та порівняльної простоті синтаксису. Надмірність мови Perl пов'язана з тим, що він розроблявся для найрізноманітніших додатків, що не могло не позначитися на його синтаксисі, в той час як JS спочатку розроблявся для Інтернету. На думку дуже багатьох користувачів, що не знали спочатку ні JS, ні Perl, вивчити JS набагато простіше.

2. JavaScript и Java

Мова програмування JavaScript, розроблений в компанії Netscape, Inc., не входить до складу платформи Java.

JavaScript не застосовується для створення аплетів або автономних додатків. Найчастіше JavaScript знаходиться всередині документів HTML і

забезпечує рівень взаємодії з веб-сторінками, якого неможливо досягти за допомогою простих документів HTML.

Основні відмінності Java і JavaScript:

Java є об'єктно-орієнтована мова програмування, а JavaScript - це об'єктно-орієнтована мова створення сценаріїв.

Java використовується для створення додатків, які запускаються на віртуальних машинах або в браузерах, а код JavaScript виконується тільки в браузері. Код Java необхідно скомпілювати, а код JavaScript використовується в текстовому вигляді. Для них потрібні різні модулі[14].

3. JavaScript и ASP.NET

ASP (Active Server Pages) - мова сценаріїв від Microsoft. Якщо мова ASP значно поступався JavaScript за багатьма параметрами, в першу чергу, за часом виконання сценаріїв, то з появою технології ASP.NET сизтуація змінилася[15].

Основна перевага ASP.NET полягає в можливості застосування всієї потужності інтерфейсу Windows для розробки Web-додатків. Це дозволяє розробляти сторінки зі складним інтерфейсом, не піклуючись про збереження, шифруванні та передачі змінних, використовуваних на динамічних сторінках.

ASP.NET тісно інтегрований в нову технологію NET компанії Microsoft. Головною особливістю розробки Web-додатків за допомогою ASP.NET є можливість використовувати будь-яку мову програмування середовища виконання NET. Це може бути як модифікований варіант Visual Basic - Visual Basic.NET, так і нову мову C # або клон Java — J #[16]. Більш того, в рамках одного Web-додатки можуть бути використані кілька мов програмування. Все це дозволяє об'єднувати в рамках одного проекту програмістів, які володіють різними мовами програмування. Код Web-додатки на ASP.NET компілюється в проміжний мова MSIL (Microsoft Intermediate Language), іноді просто званий IL. Апаратно-незалежний код на мові IL поміщається в кеш Web-сервера для підвищення продуктивності, по аналоги з байт-кодом Java. Але на відміну від технології Java, що забезпечує платформену незалежність, NET забезпечує

незалежність від мови розробки. Виконання Web-додатків ASP.NET на серверах, що працюють не під Windows, практично неможливо. Більш того, якщо Web-додаток інтенсивно використовує компоненти NET, перегляд динамічної сторінки буде можливий тільки під Windows і тільки за допомогою браузера Internet Explorer. Тому в сенсі переносимості JavaScript, безумовно, відрізняється від ASP.NET в кращу сторону. Інший вкрай важливий аргумент на користь JavaScript полягає в швидкому виправленні виявлених помилок. Теоретично виправити їх можна самостійно, якщо володієте достатньою кваліфікацією. Також можна написати лист розробникам, і з великою часткою ймовірності, в наступній версії помилка буде виправлена. А версії JavaScript виходять дуже часто. Ще одним плюсом JavaScript, особливо важливим для початківців, є те, що його легше використовувати для побудови Web-додатків.

Таким чином, при виборі між ASP.NET і JavaScript слід виходити з потреб завдання. Якщо в мережі Інтранет великої корпорації більшість серверів і клієнтських машин працюють під управлінням Windows, застосування ASP.NET більш ніж виправдано. У той же час для розробки Web-сайту, що розміщуєтьсяв Інтернеті, доцільніше використовувати JavaScript.

4. JavaScript і ColdFusion

Пакет ColdFusion розроблений фірмою Allaire і призначений для швидкої розробки як інтерактивних, так і динамічних Web-документів шляхом обробки інформації, отриманої з бази даних. Мінусом ColdFusion є досить низька переносимість. JavaScript працює практично на всіх платформах, а ColdFusion - тільки на чотирьох: Win32, Solaris, HP / UX і Linux. Крім того, ColdFusion, так само як і ASP, є комерційною розробкою. ColdFusion досить мало поширений. У порівнянні з JavaScript він більш вимогливий до ресурсів. Плюсом ColdFusion є хороша інтегроване середовище розробки і, як наслідок, більш прості, ніж в JavaScript мовні конструкції і більш швидке написання коду. Правда, через таку високорівневих мов деякі завдання на ньому вирішуються складніше.

TypeScript і JavaScript

TypeScript - це набір JavaScript, це означає, що це шар навколо JS з більшою кількістю методів, і це дозволяє слідувати певному шляху розвитку(рис. 3.1).

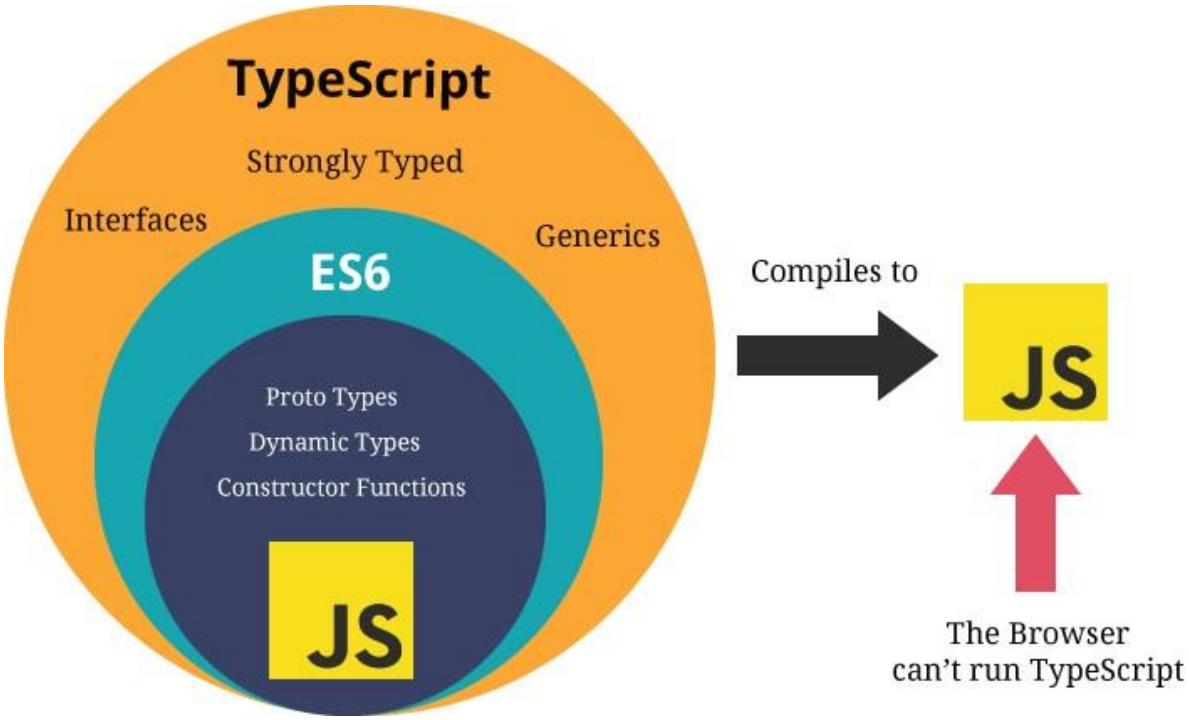


Рисунок 3.1– Зв'язок TypeScript і JavaScript

Особливості TypeScript:

1. Код TypeScript перетворюється в звичайний JS-код. Код TypeScript не зрозумілий браузерам. Ось чому, якщо код записаний у TypeScript, тоді він компілюється та перетворює код, тобто переводить код у JavaScript. Вищенаведений процес відомий як Trans-piled. За допомогою коду JavaScript браузери можуть читати код та відображати його.
2. JavaScript є TypeScript: Який би код не був написаний на JavaScript, можна перетворити в TypeScript, змінивши розширення з .js на .ts.
3. TypeScript підтримує бібліотеки JS: За допомогою TypeScript розробники можуть використовувати існуючий код JavaScript, включати популярні бібліотеки JavaScript, а також можна викликати

з іншого коду JavaScript.

Різниця між TypeScript і JavaScript:

1. TS відомий як об'єктно-орієнтована мова програмування, тоді як JavaScript - це сценарій мови.
2. TS має функцію, відому як статичне введення тексту, але JavaScript не має цієї функції.
3. Надає підтримку модулів, тоді як JavaScript не підтримує модулі.
4. Має інтерфейс, але у JavaScript немає інтерфейсу.
5. Підтримує функцію додаткового параметра, але JavaScript не підтримує функцію додаткового параметра.

Переваги використання TypeScript над JavaScript:

1. TypeScript завжди вказує на помилки компіляції лише під час розробки. Через це під час виконання шанси отримати помилки значно менші, тоді як JavaScript - це інтерпретована мова.
2. TypeScript має функцію, яка підтримує статичне введення тексту. Це означає, що статична типізація дозволяє перевірити правильність типу під час компіляції. Це недоступно в JavaScript.
3. TypeScript - це не що інше, як JavaScript та деякі додаткові функції, тобто функції ES6. Він може не підтримуватися у вашому цільовому браузері, але компілятор TypeScript може також компілювати .ts файли в ES3, ES4 та ES5.

Недоліки використання TypeScript над JavaScript:

1. Зазвичай TypeScript потребує часу для складання коду.
2. TypeScript не підтримує абстрактні класи[17].

3.2 Обґрунтування вибору середовища розробки

Для проведення аналізу засобів розробки було вибрано такі, що підтримують мову JavaScript як базову для програмування. Для проведення

аналізу було розглянуто такі інтегровані середовища розробки: Atom, WebStorm, VisualCode, Sublime Text 3.

Atom - розроблений компанією «GitHub» вільний текстовий редактор і редактор коду, який може використовуватися як самодостатнє рішення, так і у ролі технологічного стека для побудови різних спеціалізованих рішень. Зокрема, на платформі Atom побудовані середовище розробки «Visual Studio Code» від компанії «Microsoft» і «Nuclide» від «Facebook». Atom надає засоби крос-платформового редагування коду, включає вбудований пакетний менеджер і інтерфейс навігації файловою системою, надає засоби для одночасної спільної роботи з кодом, має інтелектуальну систему автодоповнення вводу, надає режими сумісності з Vim і Emacs, підтримує API для розробки розширень. Кілька файлів можуть бути відкриті в різних вкладках і одночасно відображені з використанням вертикального або горизонтального розбиття панелей. Інтерфейс може налаштовуватися через теми оформлення, підтримуються вкладки, закладки, розумний контекстний пошук коду, схлопування блоків коду, одночасне використання декількох курсорів і областей виділення, наочна позначка змін, автодоповнення та перевірка коду для різних мов (Ruby, Python, SQL, PHP, Perl, Objective-C, C/C++, JavaScript, Java, Go тощо). Для формування статей та документації може бути використана розмітка Markdown.

Visual Studio Code - засіб для створення, редагування та зневадження сучасних веб-застосунків і програм для хмарних систем. Розповсюджується безкоштовно і доступний у версіях для платформ Windows, Linux і OS X. За основу для Visual Studio Code використовуються напрацювання вільного проекту Atom, що розвивається компанією GitHub. Зокрема, Visual Studio Code є надбудовою над Atom Shell, що використовують браузерний рушій Chromium і Node.js. Примітно, що про використання напрацювань вільного проекту Atom на сайті Visual Studio Code і в прес-релізі і в офіційному блозі не згадується.[18] Редактор містить вбудований зневаджувач, інструменти для роботи з Git і засоби рефакторингу, навігації по коду, автодоповнення типових конструкцій і

контекстної підказки. Продукт підтримує розробку для платформ ASP.NET і Node.js, і позиціонується як легковаге рішення, що дозволяє обійтися без повного інтегрованого середовища розробки.

WebStorm - розроблена на основі платформи IntelliJ IDEA. WebStorm є спеціалізованою версією PhpStorm, пропонуючи підмножину з його можливостей. WebStorm постачається з перед-установленим плагінами JavaScript (такими як для Node.js), котрі доступні для PhpStorm безоплатно. WebStorm підтримує мови JavaScript, CoffeeScript, TypeScript та Dart.

WebStorm забезпечує автодоповнення, аналіз коду на льоту, навігацію по коду, рефакторинг, зневадження та інтеграцію з системами управління версіями. Важливою перевагою інтегрованого середовища розробки WebStorm є робота з проектами (у тому числі, рефакторинг коду JavaScript, що міститься в різних файлах і теках проекту, а також вкладеного в HTML). Підтримується множинна вкладеність (коли в документ на HTML вкладений скрипт на Javascript, в який вкладено інший код HTML, всередині якого вкладений Javascript) — в таких конструкціях підтримується коректний рефакторинг.

Sublime Text 3 - швидкий кросплатформенний редактор початкових текстів програм. Sublime Text не є вільним чи відкритим програмним забезпеченням, але деякі його плагіни розповсюджуються з вільною ліцензією, розробляються і підтримуються спільнотою розробників. Є кілька режимів екрану. Один з них включає від 1 до 4 панелей, за допомогою яких можна показувати до чотирьох файлів одночасно. Повноцінний (free modes) режим показує тільки один файл без будь-яких додаткових навколо нього меню.

Таблиця 3.2 – Варіантний аналіз середовищ розробки

Особливість	Коефіцієнт важливості	Atom	WebStorm	Visual Code	Sublime Text 3
Перевірка коду	1	1	1	1	1
Автодоповнення коду	1	1	1	1	1
Додавання пакетів	1	1	1	1	0
Безкоштовний	1	1	0	1	0
Оптимізація коду	1	1	1	1	1
Сума	5	5	4	5	3

За результатами варіантного аналізу, наведеними у таблиці 3.2, для розробки програмного продукту було обрано середовище розробки Atom, оскільки це середовище розробки має всі характеристики для вирішення поставленої задачі.

3.3 Програмна реалізація

Як відомо, на сьогоднішній день для написання складний, об’ємних проектів використовують готові фрейморки та бібліотеки. Одним з головних переваг при використанні фреймворків та бібліотек є те, що веб-додатки часто використовують стандартизовану структуру організації компонентів. Створення структури при розробці на фреймворками дуже спрощується.

Фреймворк — це програмне забезпечення, що полегшує розробку і об'єднання різних компонентів великого програмного проекту. Ця платформа підходить для створення сайтів, бізнес-додатків і веб-сервісів. Фреймворк відрізняється від бібліотеки тим, що бібліотека може бути використана в програмному продукті просто як набір підсистем близькою функціональності, не впливаючи на архітектуру основного програмного продукту і не накладаючи на неї ніяких обмежень. Фреймворк диктує правила побудови архітектури додатку,

задаючи на початковому етапі розробки поведінка за умовчанням, формуючи каркас, який потрібно буде розширювати і змінювати відповідно до зазначених вимог. Фреймворк може включати допоміжні програми, бібліотеки коду, мова сценаріїв і інше програмне забезпечення, що полегшує розробку і об'єднання різних компонентів великого програмного проекту.

Бібліотека — збірка об'єктів чи підпрограм для вирішення близьких за тематикою задач. У залежності від мови програмування бібліотеки містять об'єктні модулі чи сирцевий код та дані, допоміжні для задіяння та інтеграції нових можливостей в програмні рішення. Бібліотека може означати те саме, що модуль, або декілька модулів. З точки зору комп'ютерних наук бібліотеки діляться на статичні та динамічні.

Статичні бібліотеки — можуть бути у вигляді початкового тексту, що підключається програмістом до своєї програми на етапі написання (наприклад, для мови Fortran існує величезна кількість бібліотек для вирішення різних завдань саме в початкових текстах), або у вигляді об'єктних файлів, що приєднуються до виконуваної програми на етапі компіляції (у Microsoft Windows такі файли мають розширення .lib, у UNIX-подібніх ОС — зазвичай .a). В результаті програма включає всі необхідні функції, що робить її автономною, але збільшує розмір.

Динамічні бібліотеки — також називаються розподілюваними бібліотеками або бібліотеками, що динамічно підключаються. Це окремі файли, що надають програмі набір використовуваних функцій для завантажування на етапі виконання при зверненні програми до ОС із заявкою на виконання функції з бібліотеки. Якщо необхідна бібліотека вже завантажена в оперативну пам'ять, програма використовуватиме завантажену копію бібліотеки. Такий підхід дозволяє зекономити час і пам'ять, оскільки декілька програм використовують одну копію бібліотеки, вже завантажену в пам'ять.

Динамічні бібліотеки зберігаються зазвичай у визначеному місці й мають стандартне розширення. Наприклад, файли .library у логічному томі Libs: у

AmigaOS; у Microsoft Windows і OS/2 файли бібліотек загального користування мають розширення .dll; у UNIX-подібних ОС — зазвичай .so; у MacOS — .dylib.

При написанні програми програмістові досить вказати транслятору мови програмування (компілятору або інтерпретатору), що слід підключити таку-от бібліотеку і використовувати таку-от функцію зі вказаної бібліотеки. Ні початковий текст, ні виконуваний код функції до складу програми не входить.

Для написання додатку було використанно Node Js та фреймворк Express.

Node - програмна платформа, заснована на движку V8 (здійснює трансляцію JavaScript в машинний код), що перетворює JavaScript з вузькоспеціалізованого мови в мову загального призначення. Node.js додає можливість JavaScript взаємодіяти з пристроями введення-виведення через свій API (написаний на C ++), підключати інші зовнішні бібліотеки, написані на різних мовах, забезпечуючи виклики до них з JavaScript-коду. Node.js застосовується переважно на сервері, виконуючи роль веб-сервера, але є можливість розробляти на Node.js і десктопні віконні додатки (за допомогою NW.js, AppJS або Electron для Linux, Windows і macOS) і навіть програмувати мікроконтролери (наприклад, tessel і espruino). В основі Node.js лежить подієво-орієнтоване і асинхронне (або реактивне) програмування з неблокуючим введенням / висновком.[19]

Express - це веб-програма для Node.js, випущена як безкоштовне та відкрите програмне забезпечення під ліцензією MIT. Він призначений для створення веб-додатків та API. Він був названий фактично стандартною серверною рамкою для Node.js. [20]

Філософія Express полягає у наданні невеликих, надійних інструментів для HTTP-серверів, що робить його чудовим рішенням для односторінкових додатків, веб-сайтів, гібридів або загальнодоступних API HTTP.

Express не змушує вас використовувати якийсь конкретний механізм ORM або шаблон. Завдяки підтримці понад 14 двигунів шаблонів через Consolidate.js, ви можете швидко створити свій ідеальний фреймворк.

Express має багато переваг, які дозволяють спростити написання коду, а саме:

1. Міцна маршрутизація
2. Зосередьтеся на високій продуктивності
3. Супер високе покриття тесту
4. HTTP-помічники (перенаправлення, хешування тощо)
5. Перегляд системи, що підтримує 14+ двигунів шаблонів
6. Зміст переговорів
7. Виконується для швидкої генерації програм

3.3 Висновок

В результаті виконання розділу можна зробити наступні висновки:

1. При розробці клієнтського додатку доцільно використовувати мову програмування JS та середовище VS Code.
2. Для реалізації клієнтської частини було використано бібліотек React та Redux. Для реалізації серверної частини, самого сервісу отримання трейсів, було використано Node JS та бібліотеку Express.

4 ТЕСТУВАННЯ РОЗРОБЛЕНОГО ПРОГРАМНОГО ПРОДУКТУ

4.1 Тестування додатку

Тестування програмного забезпечення – процес дослідження якості, перевірки на відповідність вимогам.

Підходи до тестування програмного засобу [21]:

- статичне тестування;
- динамічне тестування;
- модульне тестування;
- тестування "білої скриньки";
- тестування "чорної скриньки".

Статичне тестування – аналіз вихідного коду програми без перевірки його фактичної працездатності, а також документації життєвого циклу програмного засобу. Відбувається перевірка сирців на очевидні помилки, на відповідність рекомендаціям з написання програмного коду, адекватність декомпозиції функцій, тощо.

Динамічне тестування програмного засобу полягає у його запуску із заздалегідь підготовленими вхідними даними. При виконанні кожного такого тесту збираються дані про відмови та іншу некоректну поведінку програми. Воно не може бути здійснено без запуску програмного коду програми. Якщо точніше, динамічне тестування складається з елементів:

- запуску системи або підсистеми;
- виклику необхідних функціональних елементів або модулів;
- порівняння через графічний інтерфейс користувача поведінки
- системи з очікуваним результатом поведінки.

Модульне тестування – це метод тестування програмного забезпечення, який полягає в окремому тестуванні кожного модуля коду програми. Модулем називають найменшу частину програми, яка може бути протестованою. У

процедурному програмуванні модулем вважають окрему функцію або процедуру. В об'єктно-орієнтованому програмуванні — метод.

Для модульного тестування додатку було використано бібліотеки і фреймворки: Jest і Enzyme.

Jest – це фреймворки JavaScript розроблюваний facebook з акцентом на простоті.

Enzyme - це утиліта JavaScript для тестування JavaScript для React, яка полегшує тестування результатів React Components. Також можна маніпулювати, переходити, а в чомусь імітувати час виконання з урахуванням результатів.

API Enzyme має бути інтуїтивно зрозумілим та гнучким, наслідуючи API jQuery для маніпуляцій та обходу DOM.

Модульне тестування також має ряд інших важливих переваг:

1. Покращує архітектуру додатків та ремонтопридатність.
2. Приводить до кращих API та композиторської здатності, зосереджуючи розробників на досвіді розробника (API) перед деталями впровадження.
3. Забезпечує швидкий зворотний зв'язок щодо збереження файлів, щоб повідомити, чи працювали ваші зміни. Це може замінити console.log () і натиснути навколо в інтерфейсі, щоб перевірити зміни.
4. Забезпечує чудову безпечну мережу, яка може підвищити вашу впевненість, коли прийде час додавати функції або рефактор існуючих функцій.

Наведемо приклад модульного тестування програмного компоненту кнопка(рис. 4.1):

```
describe('Button', () => {
  it('is not called when the "disabled" prop is true', () => {
    const onClick = jest.fn()
    const wrapper = shallow(<Button disabled onClick={onClick} />)

    wrapper.simulate('click')

    expect(onClick).not.toHaveBeenCalled()
  })
})
```

Рисунок 4.1 – Тестування програмного компонента

Кожен модульний тест розбитий на незалежні між собою блоки, в яких виконується тест. Після успішного виконання тесту в CLI буде виведено(рис. 4.2):

```
PASS  ./button.test.js
 ✓ 'is not called when the "disabled" prop is true (5ms)
```

Рисунок 4.2 – CLI після успішного виконання тесту

Було покрито вихідних код програми на шістдесят відсотків.

Покритя коду - міра, яка використовується при тестуванні програмного забезпечення. Вона визначається відсотком тестованого вихідного коду (сирцевий код) програми. Техніка покриття коду була однією із перших методик, винайдених для систематичного тестування програмного забезпечення.

Існує кілька різноманітних способів вимірювання покриття, основні з них:

1. Покриття операторів — чи кожен рядок коду був виконаний та відтестований;
2. Покриття умов — чи кожна гілка виконання (обчислення логічної умови) була виконана і відтестована;

3. Покриття шляхів — чи всі можливі шляхи через даний фрагмент коду були виконані та відтестовані;
4. Покриття функцій — чи кожна з функцій була виконана та відтестована;
5. Покриття вхід/вихід — чи всі виклики функцій та повернення із них були виконані та відтестовані.

Для програмного забезпечення із особливими вимогами до безпеки часто виникає потреба продемонструвати, що тестами досягається 100 % покриття для одного із критеріїв, але оскільки розроблювана система не являєтеся вимогливою до безпеки сто відсоткового покриття не було досягнено.

Для вимірювання покриття коду було використано вбудовану в фреймворк Jest технологію виміру покриття коду. Для її налаштування необхідно вказати за якими шляхами потрібно вимірювати покриття, це можливо зробити в jest.config.js.

Також задання граничних мір покриття можливо вказати в coverageThreshold. Поточна конфігурація:

1. Покриття шляхів – 60%
2. Покриття функцій – 60%
3. Покриття умов – 60%
4. Покриття ліній – 60%

Приклад файлу конфігурації:

```
module.exports = {
    testPathIgnorePatterns: [
        'dist/',
    ],
    collectCoverageFrom: [
         // Задання шляхів за якими буде здійснюватись покриття
        'src/**/*.tsx',
        'src/**/*.ts',
    ],
    coveragePathIgnorePatterns: [
        'export.js',
        'dist/',
        '/_tests_/',
    ],
    globals: {
        document: true,
        window: true,
    },
    moduleNameMapper: {
        '\\.(css|s+(a|c)ss)$': '<rootDir>/jest/jest.identity-obj-proxy-esm.js',
        'lodash-es': '<rootDir>/node_modules/lodash/index.js',
    },
    moduleFileExtensions: ['js', 'jsx', 'ts', 'tsx'],
    modulePaths: [
        '<rootDir>/src/',
    ],
    setupFilesAfterEnv: ['<rootDir>/src/testSetup.js'],
    testRegex: '(/_tests_/.*|(\\.|/)test)\\.tsx?$',
    transform: {
        '^.+\\.jsx?$': 'babel-jest',
        '^.+\\.tsx?$': 'babel-jest',
    },
    transformIgnorePatterns: [
        '<rootDir>/node_modules/(?!lodash-es)',
    ],
    snapshotSerializers: ['enzyme-to-json/serializer'],
    coverageThreshold: {
       // Задання мір покриття
      "global": {
        "branches": 60,
        "functions": 60,
        "lines": 60,
        "statements": 60
      }
    },
    watchPathIgnorePatterns: ["node_modules"],
}
```

Для детально відображення покриття файлу можна за допомогою звіт про покриття тесту, наприклад(рис 4.3):

```
157 128x      const buttonProps = isButton
158             ? {
159                 ...asProps,
160                 type,
161               }
162             : undefined
163
164 128x      return (
165             <Element
166               {...linkProps}
167               {...buttonProps}
168               aria-label={ariaLabel}
169               aria-expanded={ariaExpanded}
170               className={classes}
171               data-abc-id={abcId}
172               disabled={disabled}
173               onClick={this.handleClick}
174               ref={innerRef}
175             >
176               {Icon.create(icon, { defaultProps: { className: styles.icon, abcId: `${abcId}Icon` } })}
177               {hasContent(children, content) && <span className={styles.content}>{renderContent(children, content)}</span>}
178             </Element>
179           )
180         }
181
182         private readonly handleClick = (e: React.MouseEvent<GenericHTMLElement<T>, MouseEvent>) => {
183 17x       const { disabled } = this.props
184 17x       E if (!disabled) {
185 17x         invoke(this.props, 'onClick', e, this.props)
186           }
187         }
188       }
189
190  76x  Button.create = (<T extends ButtonTag = 'button'>() =>
191  76x    createShorthandFactory<ButtonProps<T>, React.ReactNode>(Button, content => ({
192             content,
193           }))())
```

Рисунок 4.3 – Звіт про покриття тесту

З позначень з лівого боку якого можна зрозуміти які лінії, умови не покриті а, які покриті в повній мірі.

Тестування «білої скриньки» означає виконання тестування із врахуванням внутрішньої структури програми, здебільшого перевірка коректності взаємодії компонентів.

Тестування «чорної скриньки» означає виконання тестування, націленого на корисні функції програмного продукту, і досліджується робота кожної функції в цілому на її області визначення. При цьому вичерпне тестування неможливе, функції перевіряються здебільшого на граничних та випадкових вхідних даних.

Під час тестування розробленого додатку для трасування запитів виконувалось тестування способом модульне тестування та чорного ящика, оскільки воно краще дає змогу віднайти помилки корисної функціональності в цілому, коли додаток уже працездатний.

Найменування об'єкта випробувань сервіс трасування запитів.

Метою попередніх комплексних випробувань є:

- перевірка працездатності програмного продукту;

- перевірка відповідності програмного продукту вимогам ТЗ;

- перевірка готовності програмного продукту до проведення дослідної комплексної експлуатації (таблиця 4.4).

Таблиця 4.4 - Тестування

Опис тесту	**Дія**	**Очікуваний результат**	**Статус**
Перевірка запуску програми	Вибір виконуваного файлу з директорії програми	У браузері з'являється вікно головної сторінки	Виконано успішно
Перевірка робити фільтрів	Встановити фільтр часу на дві години назад. Натиснути пошук.	Перенаправлення на сторінку з листом запитів	Виконано успішно
Перевірка роботи поля введення id	1. Поставити курсор на поле 2. Ввести валідний id	Перенаправлення на сторінку з розширеним описом запиту	Виконано успішно

4.2 Тестовий приклад роботи програмних модулів та аналіз результатів

Булу побудовано частини інтерфейсу веб-сервісу головна сторінка, сторінка пошуку, сторінка запиту(рис. 4.5).

Головна сторінка

Містить навігаційну частину, форму та поле для введення id трейсу. На сторінці присутні фільтри для швидкого знаходження трейсів та робоча область в який у разі знаходження запиту, буде відображена інформація про нього.(рис. 4.2.1).

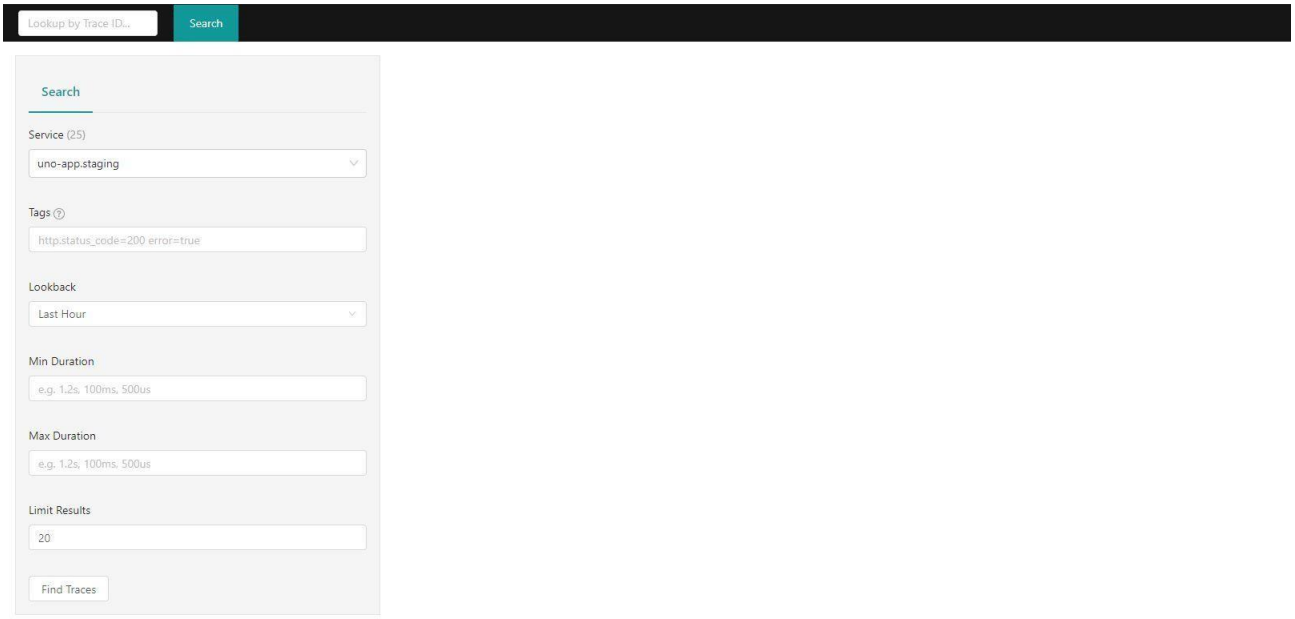


Рисунок 4.5 – Головне вікно програми

Сторінка пошуку

Головне вікно з результатами пошуку, результати відображаються на робочій частині головної сторінки. Після вдалого знаходження буде відображено лист запитів(рис. 4.6).

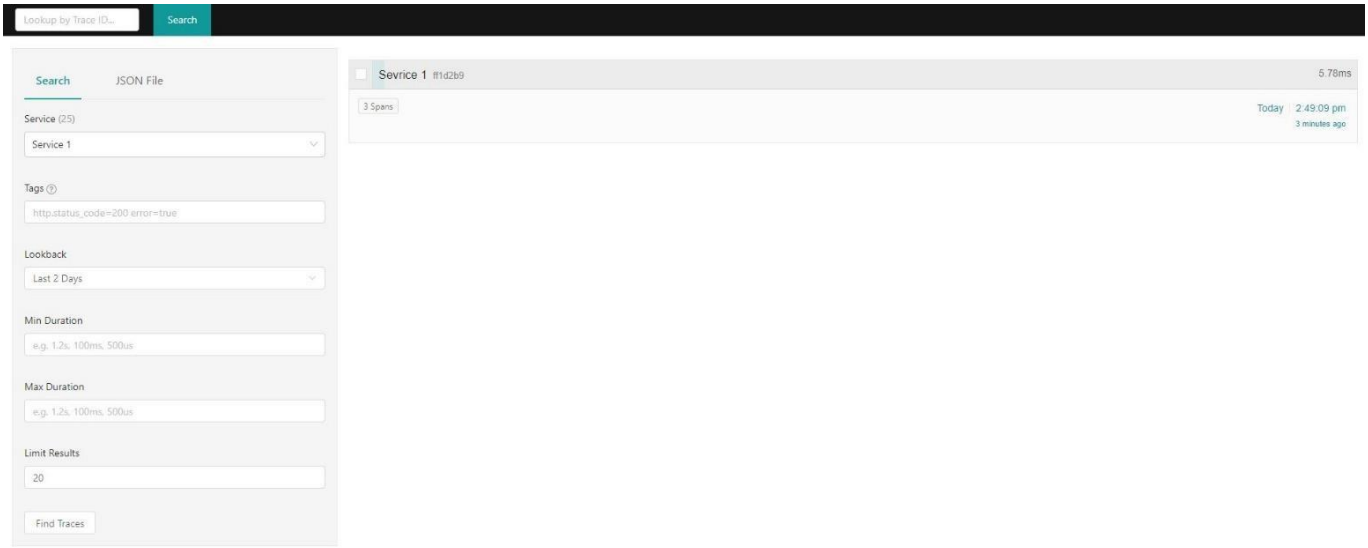


Рисунок 4.6 – Головне вікно програми з результатами пошуку

Сторінка запиту

На сторінці запиту представлено детальну інформацію про проходження запиту через мікросервіси.(рис. 4.7).

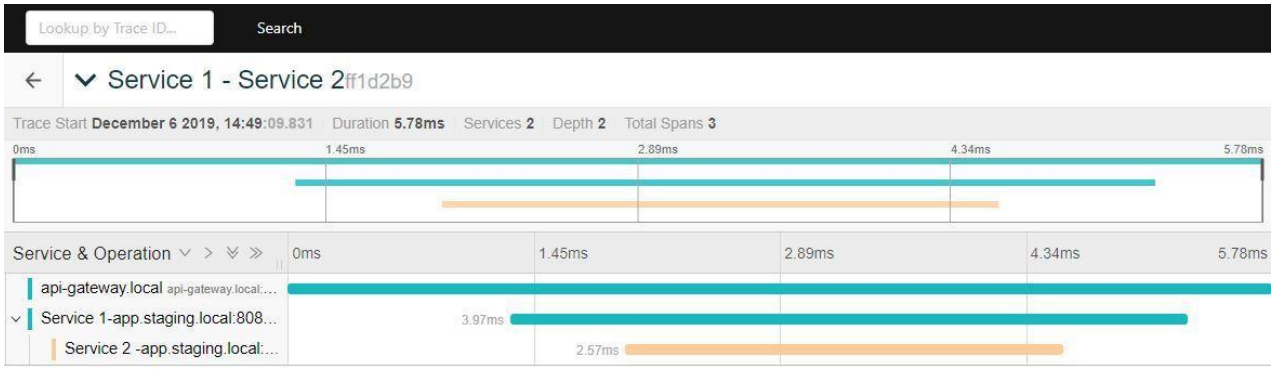


Рисунок 4.7 – Сторінка запиту

4.3 Висновок

В даному розділі було описано спосіб побудови сервісу для моніторингу запитів в додатку на основі архітектури мікросервісів, визначено технології для побудови клієнтської та серверної частини.

Також було описанно методи та способи тестування сервісу. Описанно методи та способи вимірювання кокриття коду.

Було продемострованно головні сторінки програми:

1. Головна сторінка
2. Сторінка пошуку
3. Сторінка запиту

5 ЕКОНОМІЧНА ЧАСТИНА

5.1 Оцінювання комерційного потенціалу розробки

Метою проведення технологічного аудиту є оцінювання комерційного потенціалу розробки. Для проведення технологічного аудиту було залучено 2-х незалежних експертів. Такими експертами будуть Ракитянська Г.Б та Крилик Л.В.

Здійснюємо оцінювання комерційного потенціалу розробки за 12-ма критеріями за 5-ти бальною шкалою.

Результати оцінювання комерційного потенціалу розробки наведено в таблиці 5.1.

Таблиця 5.1 – Результати оцінювання комерційного потенціалу розробки

Критерії	Прізвище, ініціали, посада експерта	
	1. Експерт 1	2. Експерт 2
	Бали, виставлені експертами:	
1	4	4
2	3	3
3	3	4
4	4	4
5	3	3
6	3	4
7	4	3
8	3	4
9	4	4
10	4	4
11	3	4
12	3	4
Сума балів	$СБ_1 = 42$	$СБ_2 = 45$
Середньоарифметична сума балів $\overline{СБ}$	$\overline{СБ} = \frac{\sum_{1}^{3} СБ_i}{2} = 43{,}5$	

Отже, з отриманих даних таблиці 5.1 видно, що нова розробка має високий рівень комерційного потенціалу.

5.2 Прогнозування витрат на виконання науково-дослідної роботи та конструкторсько–технологічної роботи.

Для розробки нового програмного продукту необхідні такі витрати.

Основна заробітна плата для розробників визначається за формулою (5.1):

$$З_o = \frac{M}{T_р} \cdot t, \qquad (5.1)$$

де М- місячний посадовий оклад конкретного розробника;

$T_р$ - кількість робочих днів у місяці, $T_р$ = 21 день;

t - число днів роботи розробника, t = 40 днів.

Розрахунки заробітних плат для керівника і програміста наведені в таблиці 5.2.

Таблиця 5.2 – Розрахунки основної заробітної плати

Працівник	Оклад М, грн.	Оплата за робочий день, грн.	Число днів роботи, t	Витрати на оплату праці, грн.
Науковий керівник	6500	309,52	8	2476,16
Інженер-програміст	4000	190,47	40	7618,8
Всього:				10094,16

Розрахуємо додаткову заробітну плату:

$$Здод = 0{,}1 \cdot 10094{,}16 = 1009{,}41 \text{ (грн.)}$$

Нарахування на заробітну плату операторів НЗП розраховується як 37,5…40% від суми їхньої основної та додаткової заробітної плати:

$$Нзп = (Зо + Зр) \cdot \frac{\beta}{100}, \qquad (5.2)$$

$$Нзп = (10094{,}16 + 1009{,}41) \cdot \frac{36{,}3}{100} = 4030{,}88 \text{ (грн.).}$$

Розрахунок амортизаційних витрат для програмного забезпечення виконується за такою формулою:

$$А = \frac{Ц \cdot Н_а}{100} \cdot \frac{Т}{12}, \quad (5.3)$$

де Ц – балансова вартість обладнання, грн;

$Н_а$ – річна норма амортизаційних відрахувань % (для програмного забезпечення 25%);

Т – Термін використання (Т=3 міс.).

Таблиця 5.3 – Розрахунок амортизаційних відрахувань

Найменування програмного забезпечення	Балансова вартість, грн.	Норма амортизації, %	Термін використання, міс.	Величина амортизаційних відрахувань,грн
Персональний комп’ютер	9000	25	3	562,5
Всього:				562,5

Розрахуємо витрати на комплектуючі. Витрати на комплектуючі розрахуємо за формулою:

$$K = \sum_1^n Н_i \cdot Ц_i \cdot К_i, \quad (5.4)$$

де n – кількість комплектуючих;

Ні - кількість комплектуючих і-го виду;

Ці – покупна ціна комплектуючих і-го виду, грн;

Кі – коефіцієнт транспортних витрат (приймемо Кі = 1,1).

Таблиця 5.4 - Витрати на комплектуючі, що були використані для розробки ПЗ.

Найменування матеріалу	Одиниці виміру	Ціна, грн.	Витрачено	Вартість витрачених матеріалів, грн.
Флешка	шт.	150	1	150
Пачка паперу	уп.	100	1	100
Ручка	шт.	5	1	5
Всього з урахуванням транспортних витрат				280,5

Витрати на силову електроенергію розраховуються за формулою:

$$B_e = B \cdot П \cdot Ф \cdot К_п; \qquad (5.5)$$

де В – вартість 1кВт-години електроенергії (В=1,7 грн/кВт);

П – установлена потужність комп'ютера (П=0,6кВт);

Ф – фактична кількість годин роботи комп'ютера (Ф=195 год.);

$К_п$ – коефіцієнт використання потужності ($К_п < 1$, $К_п = 0,8$).

$$Ве = 1,7 \cdot 0,6 \cdot 195 \cdot 0,8 = 159,12 \text{ (грн.)}$$

Розрахуємо інші витрати $В_{ін}$.

Інші витрати $І_в$ можна прийняти як (100…300)% від суми основної заробітної плати розробників та робітників, які були виконували дану роботу, тобто:

$$В_{ін} = (1..3) \cdot (З_о + З_р). \qquad (5.6)$$

Отже, розрахуємо інші витрати:

$$В_{ін} = 1 * (10094,16 + 1009,41) = 11103,57 \text{ (грн)}.$$

Сума всіх попередніх статей витрат дає витрати на виконання даної частини роботи:

$$В = З_о + З_д + Н_{зп} + А + К + В_е + І_в$$

В = 10094,16+1009,41+4030,88+562,5+280,5+159,12+11103,57= 27240,14 (грн.)

Розрахуємо загальну вартість наукової роботи $B_{заг}$ за формулою:

$$В_{заг} = \frac{В_{ін}}{\alpha} \quad (5.7)$$

де α – частка витрат, які безпосередньо здійснює виконавець даного етапу роботи, у відн. одиницях = 1.

$$В_{заг} = \frac{27240,14}{1} = 27240,14$$

Прогнозування загальних витрат ЗВ на виконання та впровадження результатів виконаної наукової роботи здійснюється за формулою:

$$ЗВ = \frac{В_{заг}}{\beta} \quad (5.8)$$

де β – коефіцієнт, який характеризує етап (стадію) виконання даної роботи.

Отже, розрахуємо загальні витрати:

$$ЗВ = \frac{27240,14}{0,9} = 30266,82 \text{ (грн.)}$$

5.3 Прогнозування комерційних ефектів від реалізації результатів розробки.

Спрогнозуємо отримання прибутку від реалізації результатів нашої розробки. Зростання чистого прибутку можна оцінити у теперішній вартості

грошей. Це забезпечить підприємству (організації) надходження додаткових коштів, які дозволять покращити фінансові результати діяльності .

Оцінка зростання чистого прибутку підприємства від впровадження результатів наукової розробки. У цьому випадку збільшення чистого прибутку підприємства $\Delta П_i$ для кожного із років, протягом яких очікується отримання позитивних результатів від впровадження розробки, розраховується за формулою:

$$\Delta П_i = \sum_1^n \quad (\Delta П_я \cdot N + П_я \Delta N)_i \qquad (5.9)$$

де $П_я \Delta$ – покращення основного якісного показника від впровадження результатів розробки у даному році;

N – основний кількісний показник, який визначає діяльність підприємства у даному році до впровадження результатів наукової розробки;

ΔN – покращення основного кількісного показника діяльності підприємства від впровадження результатів розробки;

$П_я$ – основний якісний показник, який визначає діяльність підприємства у даному році після впровадження результатів наукової розробки;

n – кількість років, протягом яких очікується отримання позитивних результатів від впровадження розробки.

В результаті впровадження результатів наукової розробки витрати на виготовлення програмного продукту зменшаться на 30 грн (що автоматично спричинить збільшення чистого прибутку підприємства на 30 грн), а кількість користувачів, які будуть користуватись збільшиться: протягом першого року – на 150 користувачів, протягом другого року – на 100 користувачів, протягом третього року – 50 користувачів. Реалізація програмного продукту до впровадження результатів наукової розробки складала 1000 користувачів, а прибуток, що отримував розробник до впровадження результатів наукової розробки – 300 грн.

Спрогнозуємо збільшення чистого прибутку від впровадження результатів наукової розробки у кожному році відносно базового.

Отже, збільшення чистого продукту $^{\Delta П_1}$ протягом першого року складатиме:

$$\Delta П_1 = 30 \cdot 1000 + (300 + 30) \cdot 150 = 79500 \text{ (грн)}.$$

Протягом другого року:

$$\Delta П_2 = 30 \cdot 1000 + (300 + 30) \cdot (150 + 100) = 112500 \text{ (грн)}.$$

Протягом третього року:

$$\Delta П_3 = 30 \cdot 1000 + (300 + 30) \cdot (150 + 100 + 50) = 129000 \text{ (грн)}.$$

5.4 Розрахунок ефективності вкладених інвестицій та період їх окупності

Визначимо абсолютну і відносну ефективність вкладених інвестором інвестицій та розрахуємо термін окупності.

Абсолютна ефективність $Е_{абс}$вкладених інвестицій розраховується за формулою:

$$Е_{абс} = (ПП - PV), \qquad (5.10)$$

де $^{\Delta П_i}$ – збільшення чистого прибутку у кожному із років, протягом яких виявляються результати виконаної та впровадженої НДДКР, грн;

т – період часу, протягом якого виявляються результати впровадженої НДДКР, 3 роки;

τ – ставка дисконтування, за яку можна взяти щорічний прогнозований рівень інфляції в країні; для України цей показник знаходиться на рівні 0,1;

t – період часу (в роках) від моменту отримання чистого прибутку до точки 2, 3,4.

Рисунок, що характеризує рух платежів (інвестицій та додаткових прибутків) буде мати вигляд, рисунок 5.1.

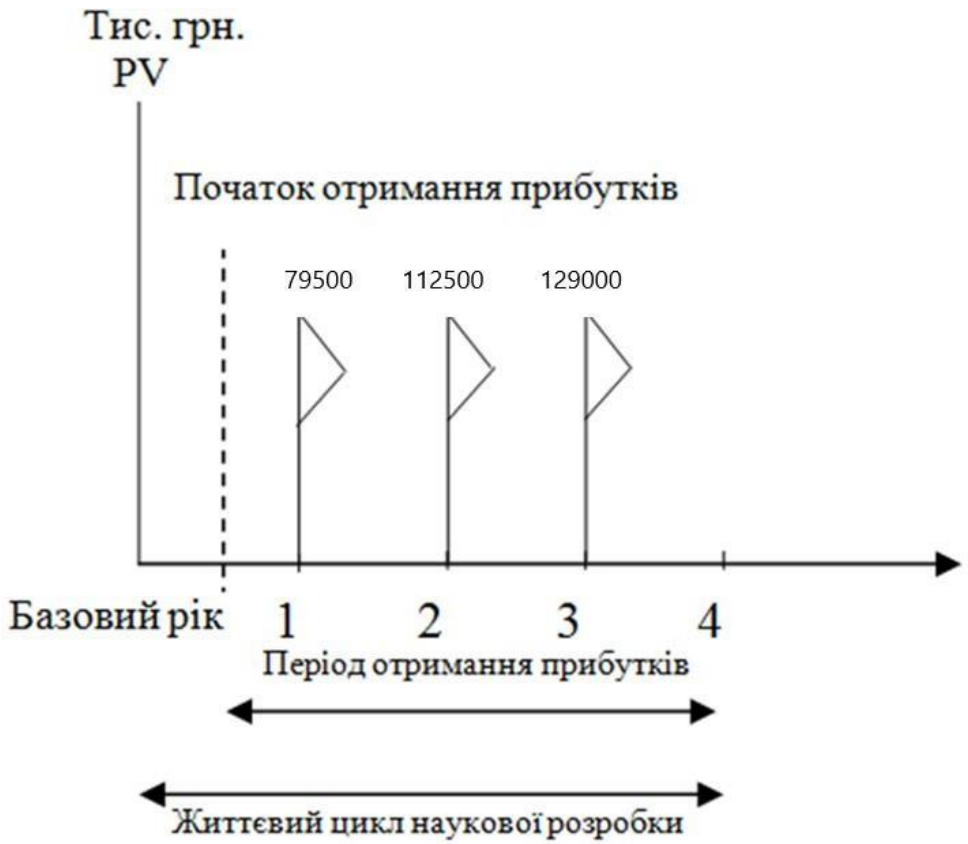


Рисунок 5.1 – Вісь часу з фіксацією платежів, що мають місце під час розробки та впровадження результатів НДДКР

Розрахуємо вартість чистих прибутків за формулою:

$$ПП = \sum_{1}^{m} \frac{\Delta П_i}{(1+\tau)^t} \quad (5.11)$$

де $\Delta П_i$ – збільшення чистого прибутку у кожному із років, протягом яких виявляються результати виконаної та впровадженої НДДКР, грн;

т – період часу, протягом якого виявляються результати впровадженої НДДКР, роки;

τ – ставка дисконтування, за яку можна взяти щорічний прогнозований рівень інфляції в країні; для України цей показник знаходиться на рівні 0,1;

t – період часу (в роках) від моменту отримання чистого прибутку до точки.

Отже, розрахуємо вартість чистого прибутку:

$$ПП = \frac{30266,82}{(1+0,1)^0} + \frac{79500}{(1+0,1)^2} + \frac{112500}{(1+0,1)^3} + \frac{129000}{(1+0,1)^4} = 268600,93 \text{ (грн.)}$$

Тоді розрахуємо $E_{абс}$:

$$E_{абс} = 268600,93 - 30266,82 = 238334,11 \text{ (грн).}$$

Оскільки $E_{абс} > 0$, то вкладання коштів на виконання та впровадження результатів НДДКР буде доцільним.

Розрахуємо відносну (щорічну) ефективність вкладених в наукову розробку інвестицій $E_в$ за формулою:

$$E_в = \sqrt[T]{1 + \frac{E_{абс}}{PV}} - 1 \qquad (5.12)$$

де $E_{абс}$ – абсолютна ефективність вкладених інвестицій, грн;

PV – теперішня вартість інвестицій PV = ЗВ, грн;

$T_ж$ – життєвий цикл наукової розробки, роки.

Тоді будемо мати:

$$E_в = \sqrt[3]{1 + \frac{238334,11}{30266,82}} - 1 = 1,07 \text{ або } 107\ \%$$

Далі, розраховану величина $E_в$ порівнюємо з мінімальною (бар'єрною) ставкою дисконтування $\tau_{мін}$, яка визначає ту мінімальну дохідність, нижче за яку інвестиції вкладатися не будуть. У загальному вигляді мінімальна (бар'єрна) ставка дисконтування $\tau_{мін}$ визначається за формулою:

$$\tau = d + f,$$

де d – середньозважена ставка за депозитними операціями в комерційних банках; в 2019 році в Україні d = 0,2;

f – показник, що характеризує ризикованість вкладень, величина f = 0,1.

$$\tau = 0{,}2 + 0{,}1 = 0{,}3$$

Оскільки $E_в = 107\% > \tau_{мін} = 0{,}3 = 30\%$, то у інвестор буде зацікавлений вкладати гроші в дану наукову розробку.

Термін окупності вкладених у реалізацію наукового проекту інвестицій. Термін окупності вкладених у реалізацію наукового проекту інвестицій $T_{ок}$ розраховується за формулою:

$$T_{ок} = \frac{1}{E_в}$$

$$T_{ок} = \frac{1}{1{,}07} = 0{,}93 \text{ року}$$

Обрахувавши термін окупності даної наукової розробки, можна зробити висновок, що фінансування даної наукової розробки буде доцільним.

ВИСНОВКИ

У магістерській кваліфікаційній роботі розроблено сервіс для моніторингу запитів. Було проаналізовано стан даної проблеми на сьогоднішній день. Сьогодні, коли більшість сучасних додатків базується на архітектурі мікросервісів, сервіс для моніторингу буде корисний.

Для того, щоб розробити додаток було проаналізовано найбільш популярні аналоги, визначено їх переваги та недоліки. Проведено варіантний аналіз засобів реалізації автоматизованої системи і обґрунтовано вибір середовища програмування та мови програмування TypeScript.

Здійснено постановку задач дослідження. Також наведено блок-схему алгоритму роботи блоку реєстрації та авторизації користувача, який належить до розгалуженого типу.

Розроблено логіку програмного продукту та розмітку вікон. Розроблено інтуїтивно-зрозумілий інтерфейс користувача та визначено, що він належить до діалогу типу меню.

Проведене тестування довело повну ефективність і правильність функціонування сервісу для моніторингу запитів.

Проведено оцінювання комерційного потенціалу розробки та проведено розрахунки, які підтверджують економічну доцільність розробки програмного продукту.

Продукт вирішує проблему відслідковування запитів між сервісами.

ПЕРЕЛІК ПОСИЛАНЬ

1. F. Chang, J. Dean, S. Ghemawat, W. C. Hsieh, D. A. Wallach, M. Burrows, T. Chandra, A. Fikes, and R. E. Gruber. Bigtable: Розподілена система зберігання структурованих даних. У працях 7-го симпозіуму USENIX щодо проектування та впровадження операційних систем. (OSDI'06), Листопад 2006. — 431 с.
2. Benjamin H. Sigelman, Luiz Andre Barroso, Mike Burrows, Pat Stephenson, Manoj Plakal, Donald Beaver, Saul Jaspan, Chandan Shanbhag. Dapper - велика масштабна інфраструктура розподілених розподілених систем. Google Technical Report dapper-2010-1, Квітень 2010. — 14 с.
3. Kraft, Johan; Wall, Anders; Kienle, Holger, "Trace Recording for Embedded Systems: Lessons Learned from Five Industrial Projects", Runtime Verification, Springer Berlin Heidelberg, 315–329.
4. Спіральная модель. URL: https://qalight.com.ua/baza-znaniy/spiralnaya-model-spiral-model.
5. И. Соммервилл. Инженерия программного обеспечения. Вильямс, 2002. — 145 с.
6. Мікросервіси та їх переваги. URL: https://internetdevels.ua/blog/building-microservices-dotnet.
7. MVC Framework – Introduction. URL: https://www.tutorialspoint.com/mvc_framework/mvc_framework_introduction.htm.
8. Addy Osmani. Learning JavaScript Design Patterns. 2008. 269 с.
9. React A JavaScript library for building user interfaces. URL: https://reactjs.org/.
10. React Fiber Architecture. URL: https://github.com/acdlite/react-fiber-architecture.
11. Redux – state container for JavaScript apps. URL: https://redux.js.org/.

12. HTTP Headers. URL: https://developer.mozilla.org/en-US/docs/Web/HTTP/Headers.
13. Date and time — Representations for information interchange. URL: https://www.iso.org/obp/ui/#iso:std:iso:8601:-2:ed-1:v1:en
14. Java Programming Language. URL: https://docs.oracle.com/javase/8/docs/technotes/guides/language/index.html.
15. ASP.net Programming Language. URL: https://dotnet.microsoft.com/apps/aspnet.
16. C# Programming Language. URL: https://docs.microsoft.com/en-us/dotnet/csharp/.
17. TypeScript Programming Language. URL: https://www.typescriptlang.org/docs/home.html.
18. VS Code. Code editor. URL: https://code.visualstudio.com/.
19. Node JS. URL: https://nodejs.org/en/about/.
20. JS – Framework. Express. URL: https://github.com/expressjs/express.
21. Тестування програмного забезпечення. URL: http://lib.mdpu.org.ua/e-book/vstup/L11.htm.

ДОДАТКИ

Додаток А. Технічне завдання

Технічне завдання на магістерську кваліфікаційну роботу

Міністерство освіти і науки України

Вінницький національний технічний університет

Факультет інформаційних технологій та комп'ютерної інженерії

ЗАТВЕРДЖУЮ

д.т.н., проф. О. Н. Романюк

"_____"_____2019 р.

Технічне завдання

на магістерську кваліфікаційну роботу за спеціальністю

121 – Інженерія програмного забезпечення

Керівник магістерської кваліфікаційної роботи:

_____к.т.н., доцент кафедри ПЗ, Ракитянська Г.Б.

"____"_____2019 р.

Виконав:

_____студент гр. 1ПІ-18м, І.Ю. Біліченко

"____"_____2019 р.

Вінниця ВНТУ 2019

1.1 Найменування та галузь застосування

Розробка сервісу для моніторингу запитів в додатках на основі архітектури мікросервісів. Згідно отриманого завдання кінцевий програмний продукт може використовуватись офіційною організацією.

1.2 Підстава для розробки

Підставою для розробки даної магістерської кваліфікаційної роботи є рішення засідання кафедри програмного забезпечення (протокол №____ від «___»_____20___року).

1.3 Мета та призначення розробки

Метою дослідження магістерської кваліфікаційної роботи є підвищення продуктивності додатку за рахунок додавання додаткових фільтрів і можливості додавання власних ключів в заголовок запиту додатку для розподіленого трасування запитів.

Для досягнення поставленої мети необхідно розв’язати такі завдання:

- провести аналіз предметної області, обґрунтування доцільності та постановка задачі розробки сервісу для моніторингу запитів в додатку на основі архітектури мікросервісів;

- здійснити проектування сервісу для моніторингу запитів в додатку на основі архітектури мікросервісів;

- розробити метод додавання власних ключів і значень в заголовок запиту;

- розробити метод сортування запитів з додаванням фільтрів з розширеними умовами пошуку;

- виконати програмну реалізацію сервісу для моніторингу запитів;

- провести тестування програмного продукту та виконати аналіз отриманих результатів.

Технічні вимоги:

- операційна система(або Браузери) – Chrome;
- мова програмування – TypeScript;
- середовище розробки – VS Code.

1.5. Перелік технічної документації, що пред’являється по закінченню робіт:

– пояснювальна записка;
– технічне обґрунтування доцільності розробки;
– лістинги програми.

1.6 Стадії і етапи розробки

Завдання на проектування видане_____2019 року. Проектування та дослідження повинно бути завершеним до_____2019 року.

Техніко-економічне обґрунтування доцільності розробки сервісу для моніторингу запитів в додатку на основі архітектури мікросервісів	
Розробка моделей і методів для моніторингу запитів	
Програмна реалізація додатку для моніторингу запитів	
Тестування роботи мобільного додатку	
Економічне обґрунтування розробки програмного продукту	
Оформлення матеріалів до захисту МКР	

1.7 Порядок контролю і приймання

Порядок контролю і приймання роботи регламентується відповідними документами ВНТУ і державними стандартами.

Додаток Б. Програмний код додатку

logic.ts

```
export class CompilationResult {
	public readonly host: fakes.CompilerHost;
	public readonly program: ts.Program | undefined;
	public readonly result: ts.EmitResult | undefined;
	public readonly options: ts.CompilerOptions;
	public readonly diagnostics: readonly ts.Diagnostic[];
	public readonly js: ReadonlyMap<string, documents.TextDocument>;
	public readonly dts: ReadonlyMap<string, documents.TextDocument>;
	public readonly maps: ReadonlyMap<string, documents.TextDocument>;
	public symlinks?: vfs.FileSet; // Location to store original symlinks so they may be used in both original and declaration file compilations

	private _inputs: documents.TextDocument[] = [];
	private _inputsAndOutputs: collections.SortedMap<string, CompilationOutput>;

	constructor(host: fakes.CompilerHost, options: ts.CompilerOptions, program: ts.Program | undefined, result: ts.EmitResult | undefined, diagnostics: readonly ts.Diagnostic[]) {
		this.host = host;
		this.program = program;
		this.result = result;
		this.diagnostics = diagnostics;
		this.options = program ? program.getCompilerOptions() : options;

		// collect outputs
		const js = this.js = new collections.SortedMap<string, documents.TextDocument>({ comparer: this.vfs.stringComparer, sort: "insertion" });
		const dts = this.dts = new collections.SortedMap<string, documents.TextDocument>({ comparer: this.vfs.stringComparer, sort: "insertion" });
		const maps = this.maps = new collections.SortedMap<string, documents.TextDocument>({ comparer: this.vfs.stringComparer, sort: "insertion" });
		for (const document of this.host.outputs) {
			if (vpath.isJavaScript(document.file) || ts.fileExtensionIs(document.file, ts.Extension.Json)) {
				js.set(document.file, document);
			}
			else if (vpath.isDeclaration(document.file)) {
				dts.set(document.file, document);
			}
			else if (vpath.isSourceMap(document.file)) {
				maps.set(document.file, document);
			}
		}

		// correlate inputs and outputs
		this._inputsAndOutputs = new collections.SortedMap<string, CompilationOutput>({ comparer: this.vfs.stringComparer, sort: "insertion" });
		if (program) {
			if (this.options.out || this.options.outFile) {
				const outFile = vpath.resolve(this.vfs.cwd(), this.options.outFile || this.options.out);
				const inputs: documents.TextDocument[] = [];
				for (const sourceFile of program.getSourceFiles()) {
					if (sourceFile) {
						const input = new documents.TextDocument(sourceFile.fileName, sourceFile.text);
						this._inputs.push(input);
						if (!vpath.isDeclaration(sourceFile.fileName)) {
							inputs.push(input);
						}
```

```
            }
        }

        const outputs: CompilationOutput = {
            inputs,
            js: js.get(outFile),
            dts: dts.get(vpath.changeExtension(outFile, ".d.ts")),
            map: maps.get(outFile + ".map")
        };

        if (outputs.js) this._inputsAndOutputs.set(outputs.js.file, outputs);
        if (outputs.dts) this._inputsAndOutputs.set(outputs.dts.file, outputs);
        if (outputs.map) this._inputsAndOutputs.set(outputs.map.file, outputs);

        for (const input of inputs) {
            this._inputsAndOutputs.set(input.file, outputs);
        }
    }
    else {
        for (const sourceFile of program.getSourceFiles()) {
            if (sourceFile) {
                const input = new documents.TextDocument(sourceFile.fileName, sourceFile.text);
                this._inputs.push(input);
                if (!vpath.isDeclaration(sourceFile.fileName)) {
                    const extname = ts.getOutputExtension(sourceFile, this.options);
                    const outputs: CompilationOutput = {
                        inputs: [input],
                        js: js.get(this.getOutputPath(sourceFile.fileName, extname)),
                        dts: dts.get(this.getOutputPath(sourceFile.fileName, ".d.ts")),
                        map: maps.get(this.getOutputPath(sourceFile.fileName, extname + ".map"))
                    };

                    this._inputsAndOutputs.set(sourceFile.fileName, outputs);
                    if (outputs.js) this._inputsAndOutputs.set(outputs.js.file, outputs);
                    if (outputs.dts) this._inputsAndOutputs.set(outputs.dts.file, outputs);
                    if (outputs.map) this._inputsAndOutputs.set(outputs.map.file, outputs);
                }
            }
        }
    }

    this.diagnostics = diagnostics;
}

public get vfs(): vfs.FileSystem {
    return this.host.vfs;
}

public get inputs(): readonly documents.TextDocument[] {
    return this._inputs;
}

public get outputs(): readonly documents.TextDocument[] {
    return this.host.outputs;
}

public get traces(): readonly string[] {
    return this.host.traces;
}
```

```
      public get emitSkipped(): boolean {
        return this.result && this.result.emitSkipped || false;
      }

      public get singleFile(): boolean {
        return !!this.options.outFile || !!this.options.out;
      }

      public get commonSourceDirectory(): string {
        const common = this.program && this.program.getCommonSourceDirectory() || "";
        return common && vpath.combine(this.vfs.cwd(), common);
      }
```

```
          public class LoginFragment extends Fragment {
          @Override
          public View onCreateView(LayoutInflater inflater, ViewGroup container, Bundle savedInstanceState) {
                    container.removeAllViews();
                    super.onCreate(savedInstanceState);
                    formView = inflater.inflate(R.layout.activity_login, null);
                    parentActivity = (MainActivity) getActivity();
                    etEmail = (EditText) formView.findViewById(R.id.email);
                    etEmail.setText(parentActivity.getCurrentUserEmail());
                    etPassword = (EditText) formView.findViewById(R.id.password);
                    etPassword.setOnEditorActionListener(new TextView.OnEditorActionListener() {
                              @Override
                              public boolean onEditorAction(TextView textView, int id, KeyEvent keyEvent) {
                                        if (id == EditorInfo.IME_NULL) {
                                                  startLogin();
                                                  return true;
                                        }
                                        return false;
                              }
                    });
                    etEmail.setText(" ");
                    etPassword.setText(" ");
                    formView.findViewById(R.id.sign_in_button).setOnClickListener(new View.OnClickListener() {
                              @Override
                              public void onClick(View view) {
                                        startLogin();
                              }
                    });
                    // If there was an error logging in.
                    if (getArguments() != null) {
                              if (getArguments().getInt("error") == Error.InvalidPassword.ordinal()) {
                                        etPassword.setError(getString(R.string.error_incorrect_password));
                                        etPassword.requestFocus();
                              } else if (getArguments().getInt("error") == Error.InvalidEmail.ordinal()) {
                                        etEmail.setError(getString(R.string.error_invalid_email));
                                        etEmail.requestFocus();
                              }
                    }
                    return formView;
          }
```

```
export default class ConstSampler extends LegacySamplerV1Base implements LegacySamplerV1 {
  _decision: boolean;

  constructor(decision: boolean) {
    super('ConstSampler');
    this._decision = decision;
  }
```

```
  name(): string {
    return 'ConstSampler';
  }

  toString(): string {
    return `${this.name()}(${this._decision ? 'always' : 'never'})`;
  }

  get decision(): boolean {
    return this._decision;
  }

  isSampled(operation: string, tags: any): boolean {
    if (this._decision) {
      tags[constants.SAMPLER_TYPE_TAG_KEY] = constants.SAMPLER_TYPE_CONST;
      tags[constants.SAMPLER_PARAM_TAG_KEY] = this._decision;
    }
    return this._decision;
  }

  equal(other: LegacySamplerV1): boolean {
    if (!(other instanceof ConstSampler)) {
    return false;
    }

    return this.decision === other.decision;
  }
}
```

```
	private void startLogin() {
		boolean cancel = false;
		View focusView = null;

		// Store values at the time of the login attempt.
		String mEmail = etEmail.getText().toString();
		String mPassword = etPassword.getText().toString();

		// Check for a valid password.
		if (TextUtils.isEmpty(mPassword)) {
			etPassword.setError(getString(R.string.error_field_required));
			focusView = etPassword;
			cancel = true;
		} else if (mPassword.length() < 4) {
			etPassword.setError(getString(R.string.error_invalid_password));
			focusView = etPassword;
			cancel = true;
		}
		// Check for a valid email address.
		if (TextUtils.isEmpty(mEmail)) {
			etEmail.setError(getString(R.string.error_field_required));
			focusView = etEmail;
			cancel = true;
		} else if (!mEmail.contains("@")) {
			etEmail.setError(getString(R.string.error_invalid_email));
			focusView = etEmail;
			cancel = true;
		}
		if (cancel) {
			// There was an error; don't attempt login and focus the first
			// form field with an error.
			focusView.requestFocus();
```

```
                } else {
                        parentActivity.setCurrentUserEmail(mEmail);
                        parentActivity.setCurrentUserPassword(mPassword);
                        parentActivity.attemptLogin();
                }
        }
}
```

trace.ts

```
export default class PrioritySampler extends BaseSamplerV2 {
  _delegates: Array<Sampler>;

  constructor(samplers: Array<Sampler | LegacySamplerV1>) {
    super('PrioritySampler');
    this._delegates = samplers.map(s => adaptSamplerOrThrow(s));
  }

  _getOrCreateState(span: Span): PrioritySamplerState {
    const store = span.context()._samplingState.extendedState();
    const stateKey = this.uniqueName();
    let state: ?PrioritySamplerState = store[stateKey];
    if (!state) {
      state = new PrioritySamplerState(this._delegates.length);
      store[stateKey] = state;
    }
    return state;
  }

  _trySampling(span: Span, fn: Function): SamplingDecision {
    const state = this._getOrCreateState(span);
    let retryable = false;
    for (let i = 0; i < this._delegates.length; i++) {
      if (state.samplerFired[i]) {
        continue;
      }
      const d = fn(this._delegates[i]);
      retryable = retryable || d.retryable;
      if (!d.retryable) {
      state.samplerFired[i] = true;
      }
      if (d.sample) {
        return d; // TODO do we want to alter out tags?
      }
    }
    return { sample: false, retryable: retryable, tags: null };
  }

  onCreateSpan(span: Span): SamplingDecision {
    return this._trySampling(span, function(delegate: Sampler): SamplingDecision {
      return delegate.onCreateSpan(span);
    });
  }

  onSetOperationName(span: Span, operationName: string): SamplingDecision {
    return this._trySampling(span, function(delegate: Sampler): SamplingDecision {
    return delegate.onSetOperationName(span, operationName);
    });
  }

  onSetTag(span: Span, key: string, value: any): SamplingDecision {
    return this._trySampling(span, function(delegate: Sampler): SamplingDecision {
      return delegate.onSetTag(span, key, value);
    });
  }

  close(callback: ?() => void): void {
```

```
    const countdownCallback = Utils.countdownCallback(this._delegates.length, callback);
    this._delegates.forEach(r => r.close(countdownCallback));
  }
}
```

span.js

```
import BaggageSetter from './baggage/baggage_setter';
import { Tags as otTags } from 'opentracing';
import SpanContext from './span_context';
import Tracer from './tracer';
import Utils from './util';

export default class Span {
  _tracer: Tracer;
  _operationName: string;
  _spanContext: SpanContext;
  _startTime: number;
  _logger: any;
  _duration: number;
  _logs: Array<LogData>;
  _tags: Array<Tag>;
  static _baggageHeaderCache: {};
  _references: Array<Reference>;
  _baggageSetter: BaggageSetter;

  constructor(
    tracer: Tracer,
    operationName: string,
    spanContext: SpanContext,
    startTime: number,
    references: ?Array<Reference>
  ) {
    this._tracer = tracer;
    this._operationName = operationName;
    this._spanContext = spanContext;
    this._startTime = startTime;
    this._logger = tracer._logger;
    this._references = references || [];
    this._baggageSetter = tracer._baggageSetter;
    this._logs = [];
    this._tags = [];
  }

  get operationName(): string {
    return this._operationName;
  }

  get serviceName(): string {
    return this._tracer._serviceName;
  }

  static  _getBaggageHeaderCache()  {
    if (!Span._baggageHeaderCache)  {
    Span._baggageHeaderCache = {};
    }

    return Span._baggageHeaderCache;
  }

  /**
   * Returns a normalize key.
   *
```

```
 * @param {string} key - The key to be normalized for a particular baggage value.
 * @return {string} - The normalized key (lower cased and underscores replaced, along with dashes.)
 **/
_normalizeBaggageKey(key: string) {
  let baggageHeaderCache = Span._getBaggageHeaderCache();
  if (key in baggageHeaderCache) {
    return baggageHeaderCache[key];
  }

  let normalizedKey: string = key.replace(/_/g, '-').toLowerCase();

  if (Object.keys(baggageHeaderCache).length < 100) {
    baggageHeaderCache[key] = normalizedKey;
  }

  return normalizedKey;
}

/**
 * Sets a baggage value with an associated key.
 *
 * @param {string} key - The baggage key.
 * @param {string} value - The baggage value.
 *
 * @return {Span} - returns this span.
 **/
setBaggageItem(key: string, value: string): Span {
  let normalizedKey = this._normalizeBaggageKey(key);

  // We create a new instance of the context here instead of just adding
  // another entry to the baggage dictionary. By doing so we keep the
  // baggage immutable so that it can be passed to children spans as is.
  // If it was mutable, we would have to make a copy of the dictionary
  // for every child span, which on average we expect to occur more
  // frequently than items being added to the baggage.
  this._spanContext = this._baggageSetter.setBaggage(this, normalizedKey, value);
  return this;
}

/**
 * Gets a baggage value with an associated key.
 *
 * @param {string} key - The baggage key.
 * @return {string} value - The baggage value.
 **/
getBaggageItem(key: string): string {
  let normalizedKey = this._normalizeBaggageKey(key);
  return this._spanContext.baggage[normalizedKey];
}

/**
 * Returns the span context that represents this span.
 *
 * @return {SpanContext} - Returns this span's span context.
 **/
context(): SpanContext {
  return this._spanContext;
}

/**
 *  Returns the tracer associated with this span.
```

```
 *
 * @return {Tracer} - returns the tracer associated with this span.
 **/
tracer(): Tracer {
  return this._tracer;
}

/**
 * Checks whether or not a span can be written to.
 *
 * @return {boolean} - The decision about whether this span can be written to.
 **/
_isWriteable(): boolean {
  return !this._spanContext.samplingFinalized || this._spanContext.isSampled();
}

/**
 * Sets the operation name on this given span.
 *
 * @param {string} name - The name to use for setting a span's operation name.
 * @return {Span} - returns this span.
 **/
setOperationName(operationName: string): Span {
  this._operationName = operationName;
  // We re-sample the span if it has not been finalized.
  if (!this._spanContext.samplingFinalized) {
    const decision = this.tracer()._sampler.onSetOperationName(this, operationName);
    this._applySamplingDecision(decision);
  }
  return this;
}

_applySamplingDecision(decision: SamplingDecision): void {
  if (!decision.retryable) {
  this._spanContext.finalizeSampling();
  }
  if (decision.sample) {
    this._spanContext._setSampled(true);
    if (decision.tags) {
    this._appendTags(decision.tags);
    }
  }
}

/**
 * Sets the end timestamp and finalizes Span state.
 *
 * With the exception of calls to Span.context() (which are always allowed),
 * finish() must be the last call made to any span instance, and to do
 * otherwise leads to undefined behavior.
 *
 * @param {number} finishTime
 *        Optional finish time in milliseconds as a Unix timestamp. Decimal
 *        values are supported for timestamps with sub-millisecond accuracy.
 *        If not specified, the current time (as defined by the
 *        implementation) will be used.
 */
finish(finishTime: ?number): void {
  if (this._duration !== undefined) {
    let spanInfo = `operation=${this.operationName},context=${this.context().toString()}`;
    this.tracer()._logger.error(`${spanInfo}#You can only call finish() on a span once.`);
```

```
    return;
  }

  if (this._spanContext.isSampled()) {
    let endTime = finishTime || this._tracer.now();
    this._duration = endTime - this._startTime;
    this._tracer._report(this);
  }
}

/**
 * Adds a set of tags to a span.
 *
 * @param {Object} keyValuePairs - An object with key value pairs
 * that represent tags to be added to this span.
 * @return {Span} - returns this span.
 **/
addTags(keyValuePairs: any): Span {
  const hasOwnProperty = Object.prototype.hasOwnProperty;
  // handle sampling.priority tag first as it can make the span writable
  const samplingKey = otTags.SAMPLING_PRIORITY;
  const samplingPriority = keyValuePairs[samplingKey];
  if (this._setSamplingPriority(samplingPriority)) {
  this._appendTag(samplingKey, samplingPriority);
  }
  if (this._isWriteable()) {
    for (let key in keyValuePairs) {
      if (hasOwnProperty.call(keyValuePairs, key)) {
        if (key === samplingKey) {
          continue; // this tag has already been added above
        }
        const value = keyValuePairs[key];
        this._appendTag(key, value);
        if (!this._spanContext.samplingFinalized) {
          const decision = this._tracer._sampler.onSetTag(this, key, value);
          this._applySamplingDecision(decision);
        }
      }
    }
  }
  return this;
}

/**
 * Adds a single tag to a span
 *
 * @param {string} key - The key for the tag added to this span.
 * @param {string} value - The value corresponding with the key
 * for the tag added to this span.
 * @return {Span} - returns this span.
 * */
setTag(key: string, value: any): Span {
  if (key === otTags.SAMPLING_PRIORITY && !this._setSamplingPriority(value)) {
    return this;
  }

  if (this._isWriteable()) {
    this._appendTag(key, value);
    if (!this._spanContext.samplingFinalized) {
      const decision = this._tracer._sampler.onSetTag(this, key, value);
      this._applySamplingDecision(decision);
```

```
    }
  }
  return this;
}

/**
 * Adds a log event, or payload to a span.
 *
 * @param {object} keyValuePairs
 *        An object mapping string keys to arbitrary value types. All
 *        Tracer implementations should support bool, string, and numeric
 *        value types, and some may also support Object values.
 * @param {number} timestamp
 *        An optional parameter specifying the timestamp in milliseconds
 *        since the Unix epoch. Fractional values are allowed so that
 *        timestamps with sub-millisecond accuracy can be represented. If
 *        not specified, the implementation is expected to use its notion
 *        of the current time of the call.
 * @return {Span} - returns this span.
 */
log(keyValuePairs: any, timestamp: ?number): Span {
  if (this._isWriteable()) {
    this._logs.push({
      timestamp: timestamp || this._tracer.now(),
      fields: Utils.convertObjectToTags(keyValuePairs),
    });
  }
  return this;
}

/**
 * Logs a event with an optional payload.
 *
 * @param {string} eventName - string associated with the log record
 * @param {object} [payload] - arbitrary payload object associated with the
 *         log record.
 */
logEvent(eventName: string, payload: any): void {
  this.log({
    event: eventName,
    payload: payload,
  });
}

/**
 * Returns true if the flag was updated successfully, false otherwise
 *
 * @param priority - 0 to disable sampling, 1 to enable
 * @returns {boolean} - true if the flag was updated successfully
 * @private
 */
_setSamplingPriority(priority: ?number): boolean {
  if (priority == null) {
    return false;
  }
  this._spanContext.finalizeSampling();
  if (priority > 0) {
    if (this._spanContext.isDebug()) {
      // If the span is already in debug, no need to set it again
      return false;
    }
```

```
      if (this._tracer._isDebugAllowed(this._operationName)) {
        this._spanContext._setSampled(true);
        this._spanContext._setDebug(true);
        return true;
      }
      return false;
    }
    this._spanContext._setSampled(false);
    this._spanContext._setDebug(false);
    return true;
  }

  _appendTag(key: string, value: any): void {
    this._tags.push({ key: key, value: value });
  }

  // Internal method that adds tags without verifying them.
  // TODO: consider if we want to remove duplicates when sampling tags are added twice.
  _appendTags(tags: {}): void {
    const hasOwnProperty = Object.prototype.hasOwnProperty;
    for (let key in tags) {
      if (hasOwnProperty.call(tags, key)) {
        const value = tags[key];
        this._appendTag(key, value);
      }
    }
  }
}
```

Додаток В. Ілюстративний матеріал

ІЛЮСТРАТИВНИЙ МАТЕРІАЛ ДО ЗАХИСТУ МАГІСТЕРСЬКОЇ КВАЛІФІКАЦІЙНОЇ РОБОТИ

Завідувач кафедри ПЗ, д. т. н., професор _____ О. Н. Романюк

Науковий керівник, к. т. н., доцент кафедри ПЗ_____ Г. Б. Ракитянська

Рецензент, к.т.н, доцент кафедри КН_____ Л. В. Крилик

Нормоконтроль, к. т. н., доцент кафедри ПЗ _____ Г. Б. Ракитянська

Виконавець, студент групи 1ПІ-18м _____ І.Ю. Біліченко

Слайд 1

Вінницький національний технічний університет
Факультет інформаційних технологій та комп'ютерної інженерії
Кафедра програмного забезпечення

Розробка методів та засобів для моніторингу запитів в додатках на основі мікросервісів

Студент 1ПІ-18м: Біліченко І.Ю.
Науковий керівник: к.т.н., доцент: Ракитянська Г.Б.

Слайд 2

Мета, об'єкт та предмет дослідження

- Метою роботи магістерської кваліфікаційної роботи є підвищення продуктивності додатку за рахунок додавання додаткових фільтрів і можливості додавання власних ключів в заголовок запиту додатку для розподіленого трасування запитів
- Об'єкт дослідження – дослідження є процес трасування запитів.
- Предметом дослідження дослідження є методи та засоби відслідковування запитів в додатках на основі архітектури мікросервісів.

Слайд 3

НАУКОВА НОВИЗНА

- Вперше запропоновано метод додавання власних ключів і значень в заголовок запиту при розподіленому трасуванні, особливість якого полягає у додаванні настроюваний файлів заголовку, що дає змогу масштабувати трасування запитів і, як наслідок, підвищити продуктивність додатку.

- Вперше запропоновано метод сортування запитів, особливість якого полягає у додаванні фільтрів з розширеними умовами пошуку, що дозволяє підвищити продуктивність додатку.

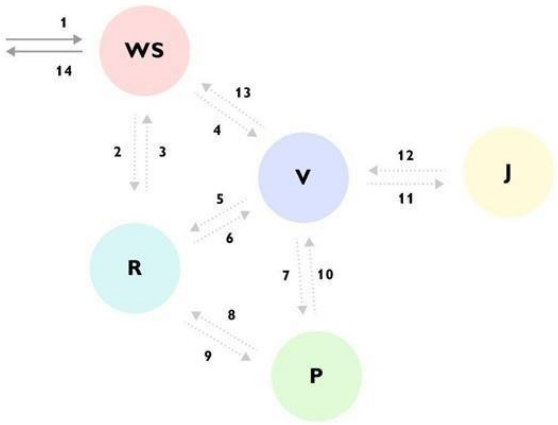
Слайд 4

Розподілений трасування

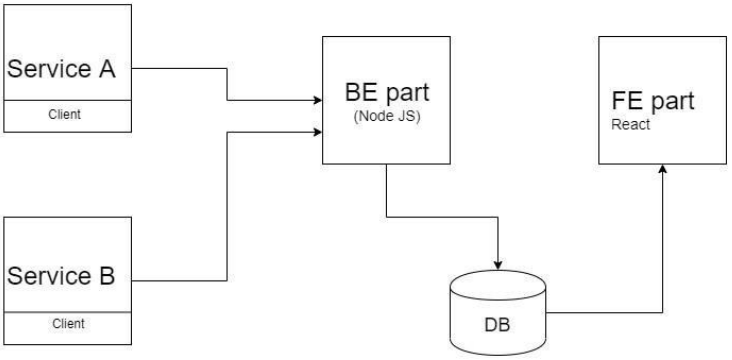
Розподілене трасування, яке також називається розподіленим відстеженням запитів, - це метод, який використовується для профілювання та моніторингу програм, особливо тих, побудованих за допомогою архітектури мікросервісів. Розподілене трасування допомагає точно визначити, де трапляються збої та що призводить до низької продуктивності.

Слайд 5

Проблеми які вирішує трасування



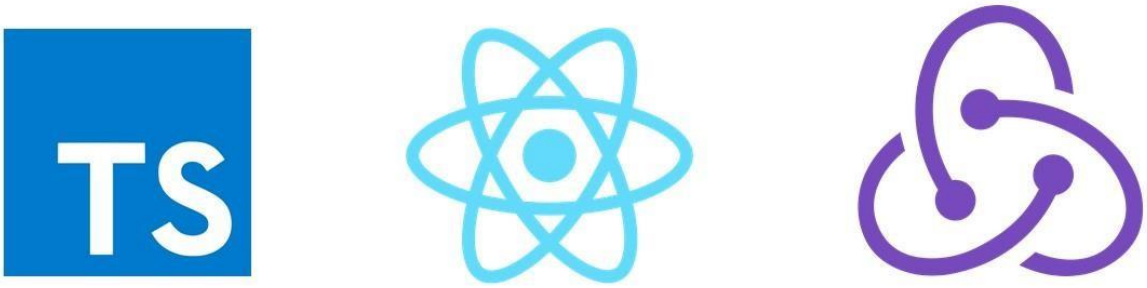
Слайд 6



Слайд 7

Використанні технології

- Мова програмування: TypeScript
- UI фреймовки: React, Redux

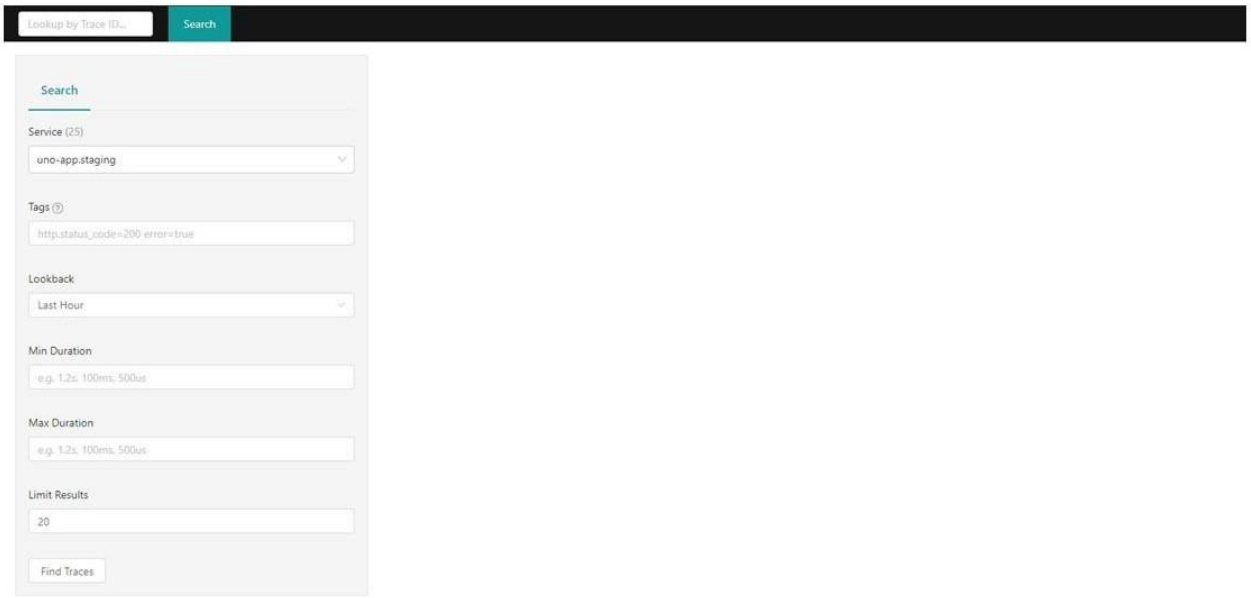


Слайд 8

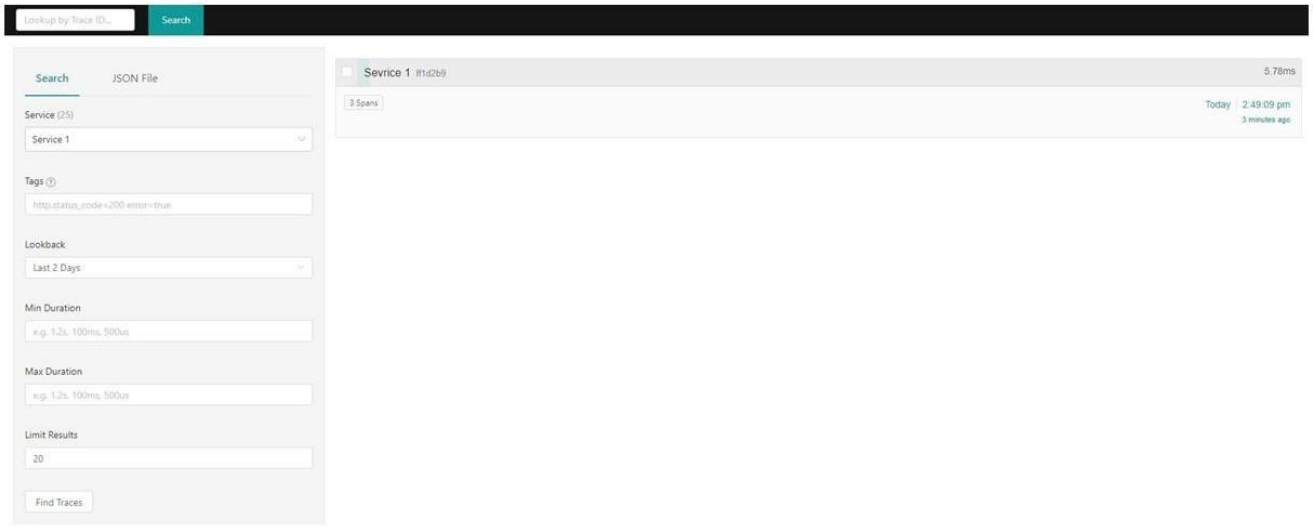
Використанні технології

- BE фрейворки: Node JS, Express
- Фреймовки для модульного тестування: Jest, Enzyme

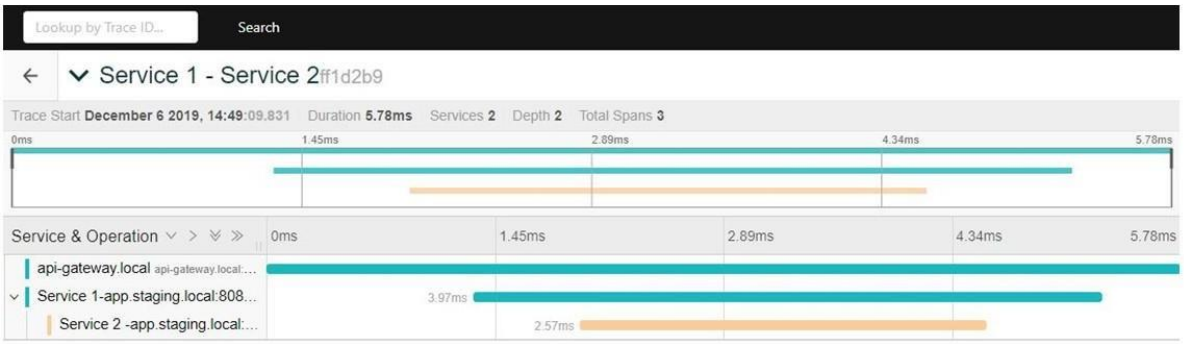
Слайд 9



Слайд 10



Слайд 11



Слайд 12

Висновки

- Розроблено сервіс для моніторингу запитів. Було проаналізовано стан даної проблеми на сьогоднішній день. Сьогодні, коли більшість сучасних додатків базується на архітектурі мікросервісів, сервіс для моніторингу буде корисний.
- Для того, щоб розробити додаток було проаналізовано найбільш популярні аналоги, визначено їх переваги та недоліки.
- Проведене тестування довело повну ефективність і правильність функціонування розробленого сервісу.