Міністерство освіти і науки України
Національний університет «Києво-Могилянська академія»
Факультет інформатики
Кафедра інформатики

МАГІСТЕРСЬКА РОБОТА НА ТЕМУ: «ФРЕЙМВОРК ТА МЕТОДОЛОГІЯ ДЛЯ ЕФЕКТИВНОГО ТЕСТУВАННЯ І ТРАСУВАННЯ У МІКРОСЕРВІСНИХ СИСТЕМАХ»

Mazurok_Manisterska_robota_1

Виконав студент 2 курсу
МП «Інженерія програмного забезпечення»
Мазурок Андрій
Науковий керівник:
д. т. н., проф. Глибовець Андрій Миколайович

2026

Актуальність теми

- **Складність архітектури:** Перехід від локальних викликів у моноліті до мережевої взаємодії мікросервісів.
- **Асинхронність:** Поширене використання брокерів повідомлень, де запит і відповідь розірвані в часі.
- **Вимоги до якості:** Потреба в надійній перевірці цілісних бізнес-сценаріїв, а не лише окремих API.
- **Трасування:** Необхідність переходу від пасивного моніторингу помилок до активного використання даних трасування для автотестів.

Проблематика E2E тестування

- **Сліпі зони в асинхронних процесах:** Важкість тестування ланцюжків, де запит і відповідь розірвані в часі.
- **Проблема очікування**: Використання фіксованих таймаутів «із запасом», що безпідставно подовжує час виконання тестів.
- **Жорстка зв'язність**: Тести прив’язані до внутрішніх баз даних чи API, що вимагає побудови додаткових інтеграційних зв’язків лише для перевірки факту виконання операції.
- **Складна локалізація помилок:** Загальні коди помилок змушують інженерів вручну аналізувати логи багатьох сервісів.

Мета та завдання дослідження

Мета: Розробити фреймворк та методологію для ефективного автоматизованого E2E-тестування мікросервісних систем на основі даних розподіленого трасування та аналізу структурованих аудит-логів.

Завдання:

1. Проаналізувати сучасні підходи до інтеграційного та E2E-тестування розподілених систем, а також існуючі стандарти наскрізного трасування.
2. Розробити методологію фіксації бізнес-операцій та створити бібліотеку логування для генерації структурованих аудит-логів у середовищі Java.
3. Забезпечити автоматичне передавання єдиного ідентифікатора транзакції через синхронні та асинхронні канали зв'язку.
4. Розробити тестовий модуль валідації життєвого циклу розподіленого запиту.
5. Інтегрувати розроблене рішення в мікросервісне середовище та підтвердити його стійкість до архітектурного рефакторингу під час тестування.

Методологія Trace-Based Тестування

- **Традиційний підхід:** Тест взаємодіє лише з точками входу та виходу системи. Внутрішні процеси приховані.
- **Trace-Based Тестування:** Тест аналізує «цифровий слід», який система генерує під час виконання запиту та перевіряє факт виконання операцій.

Ключові переваги методології:

- **Підтримка асинхронності:** Ефективна робота з асинхронними операціями – тест орієнтується на факт появи сліду події, а не на фіксований час.
- **Агностичність до архітектури:** Тест перевіряє ***бізнес-факти***, а не те, як саме сервіси поділені між собою.
- **Повна прозорість:** При падінні тесту ми бачимо повний граф виконання операції.

Архітектура розробленого рішення

Система складається з двох слабкозв'язаних компонентів:

Модуль генерації (tlog)

Прозора надбудова над SLF4J. Інкапсулює збір метаданих, серіалізацію в JSONL та асинхронний запис у фоновому потоці.

Модуль валідації (TraceAsserter)

Незалежний модуль валідації. Виконує зчитування логів, кореляцію за traceId та декларативну перевірку ланцюжка викликів.

Структура аудит-логу:

- **Ідентифікатори розподіленого трасування**: traceId, spanId.
- **Контекст операції:** timestamp, appName, source, thread, message.
- **Бізнес-дані:** operation, status.

Наскрізне трасування

Стандарт W3C Trace Context

Використання заголовка **traceparent** для єдиної ідентифікації транзакції в усій системі.

Механізм MDC (Mapped Diagnostic Context)

Збереження та автоматичне додавання ідентифікаторів до кожного рядка логу в межах Java-потоку.

Протокольна агностичність:

- **REST/HTTP:** Заголовки запиту.
- **gRPC:** Метадані виклику.
- **Apache Kafka:** Заголовки повідомлень.

Модуль автоматизованої валідації TraceAsserter

Декларативний API

Опис очікуваного ланцюжка викликів у стилі Fluent API (тести легко читаються та підтримуються).

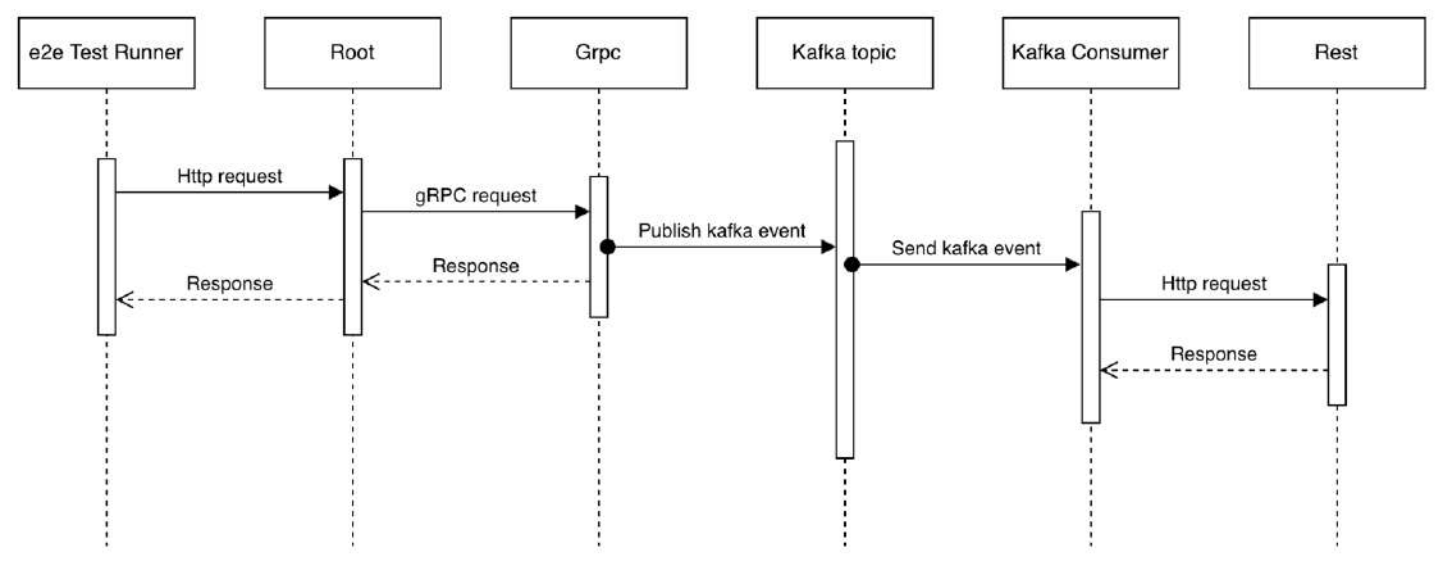
Алгоритм валідації

Асинхронний збір логів за єдиним traceId та відкладена комплексна перевірка

Багаторівневий аналіз бізнес-процесу

- **Компоненти:** Контроль того, які саме мікросервіси брали участь в обробці.
- **Етапи:** Перевірка проходження запиту через ключові класи та шари системи.
- **Операції:** Валідація виконання цільових бізнес-дій.
- **Контроль помилок:** Автоматичний аналіз системи на наявність критичних статусів в усіх залучених сервісах

Демонстраційний стенд та інфраструктура



- **Docker Compose:** Оркестрація чотирьох незалежних Java-мікросервісів для повної ізоляції тестового середовища
- **Гібридна комунікація:** Одночасне використання REST, gRPC та Apache Kafka
- **Централізований збір телеметрії:** Асинхронне агрегування JSONL логів з усіх компонентів системи у реальному часі.

Реалізація тестового сценарію

Особливості виконання:

- **Динамічний таймаут (5с):** Максимальний ліміт очікування, але тест завершується миттєво після появи останньої події.
- **Комплексна перевірка:** Одночасний контроль інфраструктури, бізнес-операцій та чистих логів (без ERROR та WARN)

```
traceAsserter.awaitForTrace(traceId, Duration.ofSeconds(5))
        .hasApps(
                "gateway",
                "order-service",
                "payment-service",
                "notification-service"
        )
        .hasJavaClasses(
                "com.shop.notification.kafka.OrderPaidConsumer"
        )
        .hasOperations(
                "CHECKOUT_REQUEST_RECEIVED",
                "ORDER_RESERVATION_COMPLETED",
                "PROCESS_PAYMENT_CALLED",
                "PAYMENT_TRANSACTION_SUCCESS",
                "NOTIFY_CLIENT_CALLED",
                "NOTIFICATION_DISPATCHED"
        )
        .completedWithoutErrors()
        .completedWithoutWarnings()
        .verify();
```

Аналіз результатів та ефективності

Критерій порівняння	Традиційний E2E підхід	Розроблене Trace-Based рішення
Час очікування результату	Фіксований (максимальний таймаут «із запасом»)	Динамічний (миттєво після появи останньої події)
Зв'язність із системою	Висока (потребує прямого доступу до БД/інтерфейсів)	Низька (абстрагована через аналіз структурованих логів)
Стійкість до рефакторингу	Низька (тести ламаються при зміні внутрішніх API)	Висока (тести стабільні, якщо збережено бізнес-логіку)
Локалізація помилок	Складна (потребує ручного збору логів при збої)	Спрощена (фреймворк відразу показує, яка операція і чому не виконалася)

Результат апробації:

- Ліквідовано надлишкові таймаути очікування в асинхронних сценаріях
- Підтверджено коректність наскрізного трасування в гібридному середовищі

Висновки

Виконання мети дослідження: Розроблено та апробовано методологію автоматизованого Trace-Based тестування розподілених систем.

Результати роботи:

- Створено Java-бібліотеку **tlog** для генерації структурованих JSONL-логів з підтримкою контексту W3C.
- Реалізовано модуль **TraceAsserter** для декларативної валідації бізнес-сценаріїв.
- Забезпечено наскрізне трасування в гібридних середовищах (REST, gRPC, Kafka).

Наукова та практична цінність:

- Скорочено час виконання асинхронних тестів за рахунок відмови від фіксованих пауз.
- Спрощено локалізацію дефектів у розподілених ланцюжках викликів.
- Доведено стійкість автотестів до архітектурного рефакторингу.

Дякую за увагу!