Java. Логування та тестування

Лекція 4

Зміст

- Логування в Java
  - Навіщо потрібне логування?
  - Рівні логування
  - Популярні фреймворки (Log4j, SLF4J, Logback)
  - Практичні приклади
- Юніт-тестування в Java
  - Що таке юніт-тести?
  - JUnit - основи
  - Assertions та Best Practices
  - Параметризовані тести
  - Mockito для мокування
- Інтеграція логування та тестування

Ситуація: Користувач скаржиться, що його платіж не пройшов, але гроші списалися з картки.

Не знаємо, що сталося

Доведеться гадати або відтворювати проблему вручну

Можливо, проблему взагалі не вдасться знайти

З логуванням

```
2025-09-25 14:23:15 INFO  [PaymentService] User 12345 initiated payment of 1000 UAH
2025-09-25 14:23:16 DEBUG [PaymentService] Calling bank API...
2025-09-25 14:23:17 INFO  [BankAPI] Payment successful, transaction ID: TXN789
2025-09-25 14:23:17 ERROR [PaymentService] Failed to update order status: DatabaseTimeout
2025-09-25 14:23:17 WARN  [PaymentService] Payment completed but order not updated for
user 12345
```

Знаходимо проблему - гроші списалися, але через `timeout` бази даних замовлення не оновилося. Швидко виправляємо статус вручну і фіксимо баг.

Вступ

Логування – це процес запису інформації про роботу програми під час її виконання.

Це структурований спосіб фіксувати події, помилки, попередження та іншу важливу інформацію, яка допомагає розробникам розуміти, що відбувається в додатку.

На відміну від `System.out.println()`, професійне логування дає можливість:

- Контролювати рівень деталізації логів
- Записувати інформацію в різні місця (файли, бази даних, консоль)
- Фільтрувати логи за рівнями важливості
- Форматувати вивід для зручного читання та аналізу

`System.out.println()` vs логування

- `println` – йде лише в консоль, зникає після перезапуску
- Логування – зберігається, структуроване, має рівні важливості, можна налаштовувати

Основні цілі

Debugging та troubleshooting - знаходження помилок у production

- Коли користувач повідомляє про помилку, логи допомагають відтворити ситуацію
- Можна побачити послідовність викликів та стан даних

Моніторинг – відстеження роботи додатку в реальному часі

- Скільки запитів обробляється
- Чи є проблеми з підключенням до БД
- Загальний стан системи

Аудит - запис важливих подій (хто, що, коли зробив)

- Фінансові операції
- Зміни в налаштуваннях системи
- Дії адміністраторів

Аналіз продуктивності - виявлення bottleneck'ів

- Які операції виконуються довго
- Де витрачається найбільше ресурсів

Рівні логування

Стандартні рівні логування (від найменш до найбільш критичного):

TRACE(600) - найдетальніша інформація, кожен крок виконання

DEBUG(500) - діагностична інформація для розробників

INFO(400) - важливі події в нормальній роботі програми

WARN(300) - попередження, щось працює не оптимально, але не критично

ERROR(200) - помилки, які потребують уваги

FATAL(100) - критичні помилки, програма не може продовжувати роботу

Числові значення визначають пріоритет рівня. Коли встановлено рівень логування (наприклад, INFO), система показує всі повідомлення з цим рівнем і вище (INFO, WARN, ERROR, FATAL). Це дозволяє фільтрувати логи за важливістю.

```
TRACE Entering method calculateDiscount() with params:
price=100, discount=10%

DEBUG Database query executed in 45ms: SELECT * FROM
users WHERE id=123

INFO User 'john@example.com' successfully logged in

WARN Database connection pool is 80% full, consider
scaling

ERROR Failed to send email to user@example.com:
SMTPException

FATAL Cannot connect to database, application shutting
down
```

```
Controller Layer:
  INFO  [UserController] POST /api/users/12345/profile - Request received from IP:
  192.168.1.100
  DEBUG [UserController] Request body: {"email":"new@example.com","phone":"+380501234567"}
Service Layer:
  INFO  [UserService] Updating profile for user ID: 12345
  DEBUG [UserService] Validating user data: email=new@example.com, phone=+380501234567
Repository/DAO Layer:
  DEBUG [UserRepository] Executing query: UPDATE users SET email=?, phone=? WHERE id=?
  TRACE [UserRepository] Query parameters: [new@example.com, +380501234567, 12345]
  DEBUG [UserRepository] Query executed successfully in 23ms
  INFO  [UserRepository] User 12345 updated successfully
Service Layer:
  DEBUG [UserService] Validation passed, user updated successfully
  INFO  [UserService] Profile update completed for user 12345
Controller Layer (Response):
  INFO  [UserController] POST /api/users/12345/profile - Success (200 OK) - Response time:
  87ms
```

Controller Layer:

```
INFO  [UserController] POST /api/users/12345/profile - Request received from IP:
192.168.1.100
DEBUG [UserController] Request body: {"email":"new@example.com","phone":"+380501234567"}
```

Service Layer:

```
INFO  [UserService] Updating profile for user ID: 12345
DEBUG [UserService] Validating user data: email=new@example.com, phone=+380501234567
```

Repository/DAO Layer:

```
DEBUG [UserRepository] Executing query: UPDATE users SET email=? WHERE id=?
ERROR [UserRepository] Database constraint violation: Duplicate entry
'existing@example.com'
```

Service Layer:

```
ERROR [UserService] Failed to update user 12345: Email already exists in
database
```

Controller Layer (Response):

```
WARN  [UserController] POST /api/users/12345/profile - Client error (409
Conflict) - Response time: 45ms
```

Популярні фреймворки

1. java.util.logging (JUL)
- Вбудований в Java з версії 1.4, не потребує додаткових залежностей
- Простий у використанні для базових потреб
- Обмежені можливості конфігурації
- Рідко використовується в enterprise-проектах через слабку функціональність
- Конфігурується через файл logging.properties

2. Log4j 2
- Повна переписана версія Log4j 1.x
- Асинхронне логування для високої продуктивності
- Гнучка конфігурація через XML, JSON, YAML або програмно
- Підтримка різних appender'ів (консоль, файли, бази даних, Kafka, тощо)
- Автоматичне перезавантаження конфігурації без перезапуску додатку
- Проблеми з безпекою в старих версіях (Log4Shell vulnerability)

3. Logback
- Нативна підтримка SLF4J
- Автоматичне стиснення та архівування старих логів
- Умовна обробка в конфігурації
- Конфігурація через XML файл (logback.xml)

4. SLF4J (Simple Logging Facade for Java)
- Це абстракція/фасад, а не реальна реалізація логування
- Дозволяє писати код незалежно від конкретного фреймворку
- Можна легко змінити Log4j на Logback без зміни коду
- Підтримує параметризовані повідомлення для кращої продуктивності
- Де-факто стандарт в Java-спільноті

- Типова зв'язка: SLF4J (API) + Logback/Log4j2 (Implementation)

```java
import java.util.logging.Level;
import java.util.logging.Logger;

nataliia
public class GroupFileParser {
    private static final Logger logger = Logger.getLogger(GroupFileParser.class.getName());

    /** Parses a single CSV line into a Group object. ...*/
    nataliia
    public static Group parseGroupFromLine(String line) throws InvalidDataException {...}

    /** Reads groups from a CSV file. ...*/
    nataliia
    public static List<Group> parseFromCSV(String filePath) throws IOException, InvalidDataException {
        List<Group> groups = new ArrayList<>();
        Path path = Path.of(filePath);

        if (!Files.exists(path)) {
            throw new IOException("File not found: " + filePath);
        }

        logger.log(Level.INFO, msg: "Starting to parse groups from file: {0}", filePath);
```

```java
import org.slf4j.LoggerFactory;
import ua.university.exception.InvalidDataException;
import ua.university.util.SubjectUtils;

nataliia *
public record Subject(String name, int credits) implements Comparable<Subject> {

    private static final Logger logger = LoggerFactory.getLogger(Subject.class);

    nataliia *
    public Subject {
        String trimmedName = name != null ? name.trim() : null;

        if (!SubjectUtils.isValidName(trimmedName)) {
            String errorMsg = "Invalid subject name: '" + name + "'";
            logger.error(errorMsg);
            throw new InvalidDataException(errorMsg);
        }

        if (!SubjectUtils.isValidCredit(credits)) {
            String errorMsg = "Invalid credit amount: " + credits + " (must be 1-5)";
            logger.error(errorMsg);
            throw new InvalidDataException(errorMsg);
        }

        name = trimmedName;
        logger.info("Subject created successfully: {} with {} credits", name, credits);
    }
```

Конфігурація

```
<configuration>
    <appender name="CONSOLE" class="ch.qos.logback.core.ConsoleAppender">
        <encoder>
            <pattern>%d{HH:mm:ss.SSS} [%thread] %-5level %logger{36} - %msg%n</pattern>
        </encoder>
    </appender>

    <appender name="FILE" class="ch.qos.logback.core.FileAppender">
        <file>app.log</file>
        <encoder>
            <pattern>%d{yyyy-MM-dd HH:mm:ss} %-5level %logger - %msg%n</pattern>
        </encoder>
    </appender>

<logger name="com.myapp" level="DEBUG"/>

    <root level="INFO">
        <appender-ref ref="CONSOLE" />
        <appender-ref ref="FILE" />
    </root>
</configuration>
```

Best practices

- Що логувати:
  - ✅ Важливі бізнес-події (створення замовлення, оплата)
  - ✅ Помилки та виключення з контекстом
  - ✅ Початок/завершення критичних операцій
  - ❌ Конфіденційні дані (паролі, токени, номери карток)
  - ❌ Надто багато `DEBUG` в `production`
- Як логувати:
  - Не дублюйте логи - якщо `Exception` логується в `Repository`, не треба логувати його знову в `Service`
  - Логуйте на правильному рівні (не все `ERROR`, не все `INFO`)
  - Додавайте контекст: не просто `"Error"`, а `"Failed to save user 12345 to database"`
- Продуктивність:
  - Для ресурсномістких операцій додавайте перевірку рівня: `if (logger.isDebugEnabled()) { logger.debug(...); }`
  - Уникайте `String concatenation` в параметрах логування. Завжди використовуйте параметризацію `logger.info("User {}", username)` замість `logger.info("User " + username)`
- Структура:
  - Включайте `timestamp`, `thread`, `logger name`, рівень
  - Використовуйте змістовні назви логерів (зазвичай ім'я класу)

Unit тестування в Java

Що таке юніт-тести?

- Юніт-тест – це автоматичний тест, який перевіряє роботу невеликої одиниці коду (зазвичай одного методу або класу) ізольовано від інших частин системи.
- Основні характеристики:
  - Тестує одну конкретну функціональність
  - Швидко виконується (мілісекунди)
  - Не залежить від зовнішніх систем (БД, API, файли)
  - Повторюваний – завжди однаковий результат
- Навіщо потрібні:
  - Впевненість, що код працює правильно
  - Легше знаходити баги
  - Безпечний рефакторинг
  - Документація коду (тести показують, як користуватися класом)

JUnit - основи

Що таке JUnit?

- Найпопулярніший фреймворк для юніт-тестування в Java
- Поточна версія - JUnit 5 (Jupiter)
- Надає анотації, assertions та інструменти для запуску тестів

Основні анотації:

- @Test - позначає метод як тестовий
- @BeforeEach - виконується перед кожним тестом
- @AfterEach - виконується після кожного тесту
- @BeforeAll - виконується один раз перед всіма тестами (статичний метод, якщо не вказано @TestInstance(PER_CLASS))
- @AfterAll - виконується один раз після всіх тестів (статичний метод)
- @DisplayName - задає зрозумілу назву тесту
- @Disabled - вимикає тест

```
class CalculatorTest {

    @Test
    void shouldAddTwoNumbers() {
        // Arrange (підготовка)
        Calculator calc = new Calculator();

        // Act (дія)
        int result = calc.add(2, 3);

        // Assert (перевірка)
        assertEquals(5, result);
    }
}
```

Assertions та Best Practices

Основні Assertions з JUnit 5:

- assertEquals(expected, actual) – перевірка рівності
- assertTrue(condition) / assertFalse(condition) – перевірка boolean
- assertNull(object) / assertNotNull(object) – перевірка на null
- assertThrows(Exception.class, () -> {...}) – перевірка виключень
- assertAll() – групування assertions

Best Practices:

- Один тест – одна перевірка (один концепт)
- Зрозумілі назви тестів (що тестується і очікуваний результат)
- Arrange-Act-Assert патерн
- Тести мають бути незалежними один від одного
- Не використовуйте логіку в тестах (if, loops)
- Додавайте повідомлення до кожного assertion

assertAll vs множинні assert

Проблема з множинними assert:

```
@Test
void testUserValidation() {
    User user = new User("John", "john@example.com", 25);

    assertEquals("John", user.getName());       // Якщо
fail - тест зупиняється тут
    assertEquals("john@example.com", user.getEmail());  //
Це не виконається
    assertEquals(25, user.getAge());            // І це теж
}
```

Якщо перша перевірка провалиться - ми не побачимо результати інших перевірок.

Рішення - assertAll():

```
@Test
void testUserValidation() {
    User user = new User("John", "john@example.com", 25);

    assertAll("User validation",
        () -> assertEquals("John", user.getName()),
        () -> assertEquals("john@example.com",
user.getEmail()),
        () -> assertEquals(25, user.getAge())
    );
}
```

Переваги assertAll:

Виконує ВСІ перевірки, навіть якщо якась провалилася

Показує всі помилки одразу

Параметризовані тести

Дозволяє запустити один тест з різними наборами даних, замість дублювання коду.

Основні джерела даних:

`@ValueSource` - прості значення одного типу:

```
@ParameterizedTest
@ValueSource(ints = {1, 2, 3, 5, 8})
void shouldBePositive(int number) {
    assertTrue(number > 0);
}
```

`@CsvSource` - кілька параметрів через кому:

```
@ParameterizedTest
@CsvSource({
    "2, 3, 5",
    "10, 5, 15",
    "-2, 2, 0"
})
void shouldAddNumbers(int a, int b, int expected) {
    assertEquals(expected, calculator.add(a, b));
}
```

`@MethodSource` - складні об'єкти через метод:

```
@ParameterizedTest
@MethodSource("provideUserData")
void shouldValidateUser(String name, String email,
boolean isValid) {
    assertEquals(isValid, validator.isValid(name,
email));
}

static Stream<Arguments> provideUserData() {
    return Stream.of(
        Arguments.of("John", "john@example.com",
true),
        Arguments.of("", "invalid", false)
    );
}
```

Заглушки

Що таке Mock?

- Імітація (заглушка) реального об'єкта для тестування
- Дозволяє ізолювати тестований клас від залежностей
- Можна контролювати поведінку та перевіряти виклики методів

Навіщо потрібно:

- Тестувати клас без реальної БД, API, файлової системи
- Симулювати різні сценарії (помилки, тайм-аути)
- Швидкі тести без зовнішніх залежностей

Основні можливості Mockito:

- Створення mock-об'єктів
- Налаштування поведінки (stubbing)
- Перевірка викликів (verification)

Приклад

```
class UserServiceTest {

    @Mock
    private Logger logger;

    @Mock
    private UserRepository userRepository;

    @InjectMocks
    private UserService userService;

    @BeforeEach
    void setUp() {
        MockitoAnnotations.openMocks(this);
    }

    @Test
    void shouldLogErrorWhenUserNotFound() {
        // Arrange
when(userRepository.findById(123)).thenReturn(null);

        // Act
        userService.getUser(123);

        // Assert - перевіряємо що ERROR лог був
викликаний
        verify(logger).error("User not found: {}",
123);
    }
}
```

@Mock - створює mock об'єкт
@InjectMocks - створює тестований об'єкт і впроваджує в нього mock'и
verify() - перевіряє, що метод був викликаний з певними параметрами

Інтеграція логування та тестування

При тестуванні:

- Використовуйте окремий конфіг логування для тестів (`test/resources/logback-test.xml`)
- Встановлюйте рівень `ERROR` або `OFF` для тестів, щоб не засмічувати консоль
- Не перевіряйте логування в кожному тесті – це не бізнес-логіка

При розробці:

- Додавайте логування на етапі написання коду, а не після
- `DEBUG` логи допомагають при написанні тестів
- Якщо тест падає – додайте логування для швидшої діагностики

Загальні поради:

- Логування та тести доповнюють один одного
- Логи допомагають знайти проблему в `production`
- Тести гарантують, що код працює правильно