

Rapport de Projet : InjectLog4J

Système d'Injection de Logs Déclaratif pour Applications Java

Auteur : Jérémy Hurel

Université de Montpellier - Master Informatique

Date : Décembre 2025

Table des Matières

1. [Contexte du Projet](#)
 2. [La Bibliothèque InjectLog4J](#)
 3. [Architecture Technique](#)
 4. [Workflow de Logging](#)
 5. [Configuration Déclarative](#)
 6. [Pipeline d'Observabilité](#)
 7. [Application de Démonstration](#)
 8. [Politique d'Utilisation de l'IA](#)
 9. [Conclusion et Perspectives](#)
 10. [Références](#)
-

1. Contexte du Projet

Ce projet a été réalisé dans le cadre d'un **TP de Master Informatique** à l'Université de Montpellier. L'objectif initial était simple : **tracer les opérations de lecture et d'écriture en base de données**.

Plutôt que de me limiter à l'ajout manuel de logs dans le code, j'ai choisi d'aller plus loin en développant **InjectLog4J**, une bibliothèque qui externalise complètement la configuration des logs. Cette approche permet de :

- Définir les règles de logging dans des fichiers YAML séparés du code
- Injecter automatiquement les logs aux bons endroits (entrée/sortie de méthode, exceptions)
- Acheminer les logs vers différentes destinations (console, fichiers, Kafka)

Le projet inclut également une infrastructure d'observabilité complète (OpenTelemetry, Jaeger, Grafana, ClickHouse) pour démontrer l'intégration des logs dans un écosystème de monitoring moderne.

2. La Bibliothèque InjectLog4J

2.1 Vue d'Ensemble

InjectLog4J est le cœur de ce projet. C'est une bibliothèque Java qui utilise **Spoon** pour transformer le code source et injecter automatiquement des appels de logging basés sur une configuration YAML.

Fonctionnalités principales :

- Configuration déclarative via fichiers YAML
- Support multi-sorties : terminal (Log4j2), Kafka, fichiers multiples
- Catégorisation des loggers (**system**, **business**)
- Déclencheurs flexibles : entrée de méthode, retour, exception
- Templates de messages avec placeholders (**{{time}}**, **{{value}}**, **{{args}}**)
- Matching par wildcards pour cibler plusieurs méthodes

2.2 Structure du Projet

```
InjectLog4J/
├── src/main/java/fr/umontpellier/injectlog4j/
│   ├── InjectLog4J.java           # Point d'entrée principal
│   ├── annotation/                # Annotations (@InjectLog)
│   ├── config/                   # Chargement configuration YAML
│   ├── processor/                 # Transformation Spoon
│   │   ├── SpoonProjectProcessor.java
│   │   ├── InjectLogProcessor.java
│   │   └── ActionAwareCodeInjector.java
│   ├── output/                   # Destinations de sortie
│   │   ├── TerminalOutput.java
│   │   ├── KafkaOutput.java
│   │   ├── FileOutput.java
│   │   └── CompositeOutput.java
│   ├── formatter/                # Formatage des messages
│   └── runtime/                  # Exécution à chaud
│       └── LogInjector.java
```

2.3 Utilisation Basique

L'intégration dans un projet Spring Boot se fait en quelques lignes :

```
@SpringBootApplication
public class MyApplication {

    public static void main(String[] args) {
        // Initialise le système de logging
        InjectLog4J.initialize();

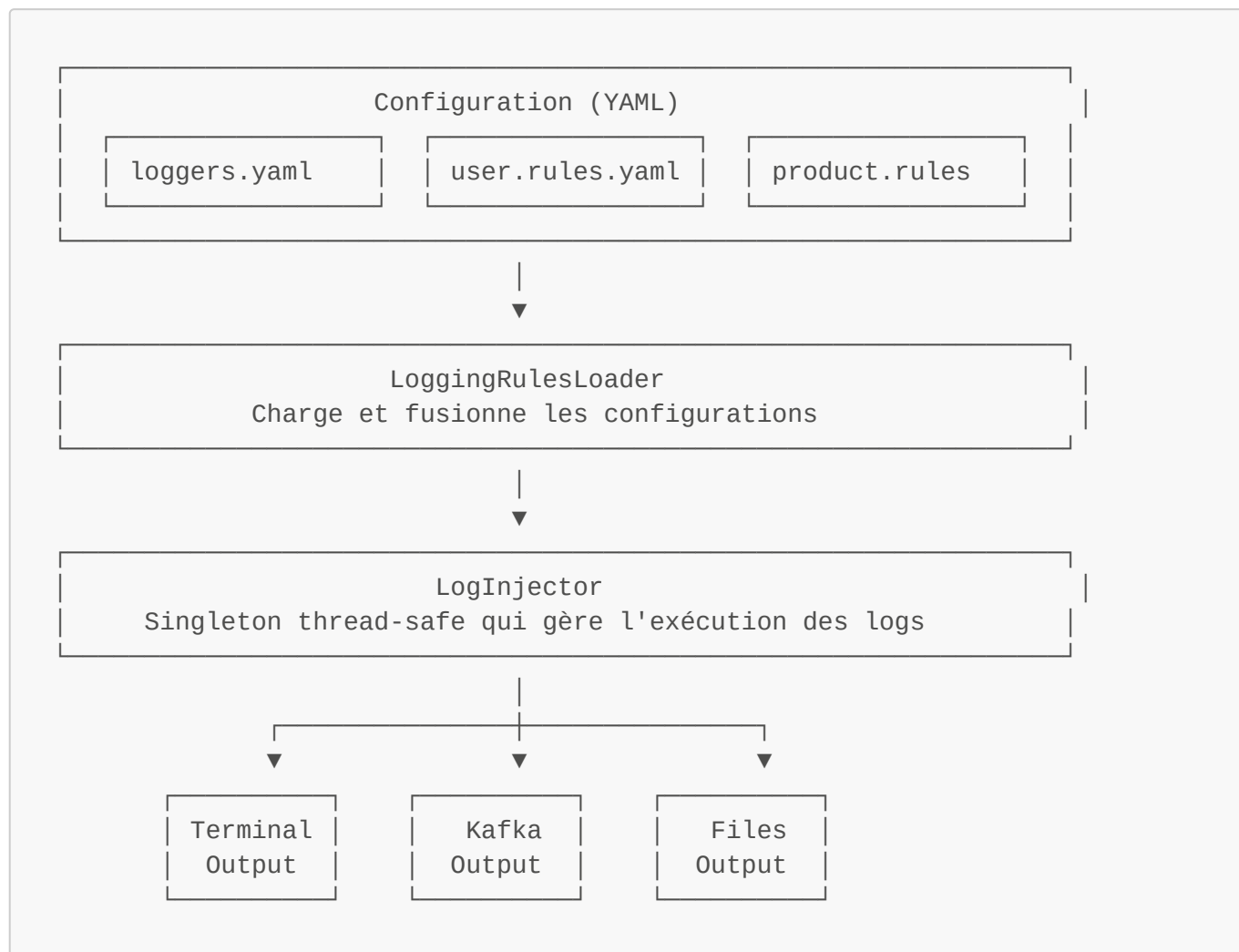
        SpringApplication.run(MyApplication.class, args);
    }

    @PreDestroy
    public void onShutdown() {
        // Libère les ressources proprement
        InjectLog4J.shutdown();
    }
}
```

3. Architecture Technique

3.1 Composants Principaux

L'architecture suit les principes SOLID et sépare clairement les responsabilités :



3.2 Le Pattern Composite pour les Sorties

Un des aspects intéressants de l'architecture est l'utilisation du pattern **Composite** pour gérer les sorties multiples. Un même log peut être envoyé simultanément vers la console, un fichier et Kafka :

```
public class CompositeOutput implements LogOutput {
    private final List<LogOutput> outputs;

    @Override
    public void write(String message, String level) {
        for (LogOutput output : outputs) {
            output.write(message, level);
        }
    }
}
```

3.3 Transformation de Code avec Spoon

La partie la plus technique du projet est la transformation de code source avec Spoon. Le `SpoonProjectProcessor` orchestre cette transformation :

```
public class SpoonProjectProcessor implements ProjectProcessor {

    @Override
    public void process(Path inputPath, Path outputPath, LoggingRulesConfig
config) {
        Launcher launcher = new Launcher();
        launcher.addInputResource(inputPath.toString());
        launcher.setSourceOutputDirectory(outputPath.toString());
        launcher.getEnvironment().setComplianceLevel(21);
        launcher.addProcessor(new InjectLogProcessor(config, new
ActionAwareCodeInjector(true)));
        launcher.run();
    }
}
```

3.4 Le LogInjector : Cœur du Runtime

Le `LogInjector` est le singleton qui gère l'exécution des logs au runtime. Il utilise le pattern "Initialization-on-demand holder" pour une initialisation thread-safe et lazy :

```
public class LogInjector {

    private static final class InstanceHolder {
        private static final LogInjector INSTANCE = createInstance();
    }

    public static LogInjector getInstance() {
        return InstanceHolder.INSTANCE;
    }

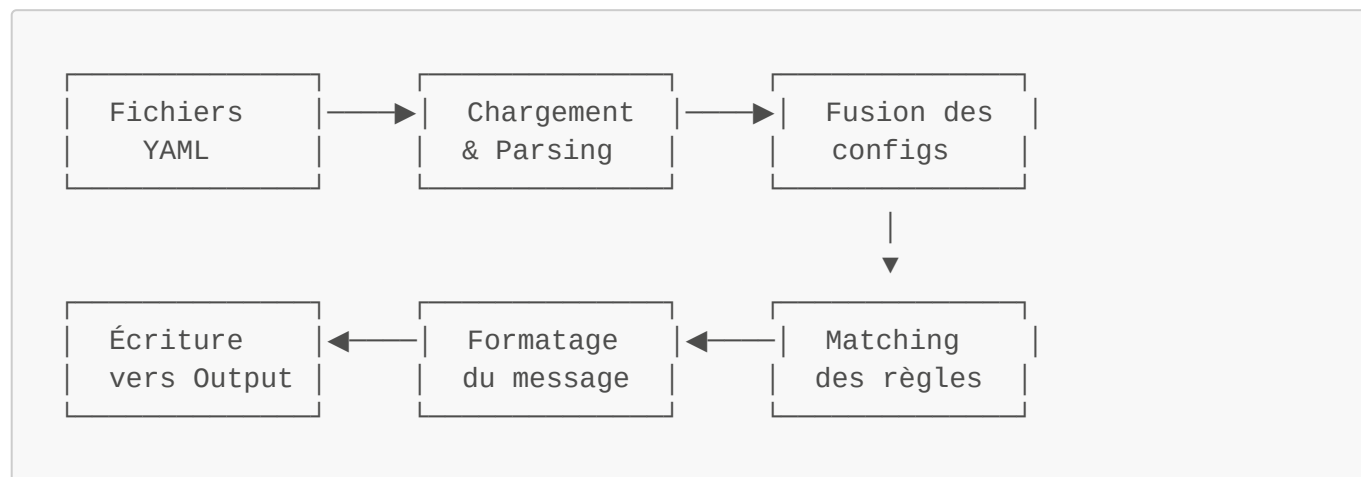
    private final Map<String, LogOutput> outputs = new ConcurrentHashMap<>
();
    private final Map<String, MessageFormatter> formatters = new
ConcurrentHashMap<>();

    // Initialise les sorties à partir de la configuration
    private void initializeOutputs() {
        for (Map.Entry<String, LoggerConfig> entry :
config.getLoggers().entrySet()) {
            outputs.put(entry.getKey(),
LogOutputFactory.create(entry.getKey(), entry.getValue()));
        }
    }
}
```

4. Workflow de Logging

4.1 Du YAML au Log

Voici comment fonctionne le workflow complet, de la configuration jusqu'à l'écriture du log :



4.2 Exemple Concret

Prenons un exemple avec la création d'un utilisateur. Voici la règle définie dans

`user.logging.rules.yaml` :

```
rules:
- target: fr.umontpellier.observability.service.UserService.createUser
  criticality: INFO
  why: [OnEntry, OnReturn]
  message: "Creating new user - result: {{value}}"
  logger: business
```

Quand la méthode `createUser` est appelée :

1. **OnEntry** : Un log est émis à l'entrée de la méthode
2. La méthode s'exécute normalement
3. **OnReturn** : Un log est émis avec la valeur de retour

Le développeur n'a rien eu à écrire dans son code, la logique métier reste propre :

```
public User createUser(User user) {
    if (userRepository.existsByEmail(user.getEmail())) {
        throw new UserAlreadyExistsException(user.getEmail());
    }
    User savedUser = userRepository.save(user);
    publishUserEvent("USER_CREATED", savedUser);
    return savedUser;
}
```

4.3 Gestion des Exceptions

La gestion des exceptions est particulièrement utile. On peut capturer les erreurs sans polluer le code :

```
rules:
  - target:
    fr.umontpellier.observability.service.UserService.publishUserEvent
    criticality: WARN
    why: [OnException]
    message: "Failed to publish user event: {{exception}}"
    logger: system
```

Si la publication Kafka échoue, le log sera automatiquement généré avec les détails de l'exception.

5. Configuration Déclarative

5.1 Définition des Loggers

Les loggers sont définis dans un fichier central `loggers.logging.rules.yaml` :

```
# Catégories de loggers :
# - system : Infrastructure, configuration, opérations techniques
# - business : Logique métier, actions utilisateur, opérations domaine

loggers:
  system:
    output: terminal
    log4jLogger: fr.umontpellier.observability.system
    format: "{{time}} [SYSTEM] [{{class}}.{{method}}] {{message}}"
    category: system
    files:
      - path: logs/system.log
        format: "{{time}} [{{level}}] [{{class}}.{{method}}] {{message}}"
        append: true
        maxSize: 10MB
        maxFiles: 5

  business:
    output: terminal
    log4jLogger: fr.umontpellier.observability.business
    format: "{{time}} [BUSINESS] [{{class}}.{{method}}] {{message}}"
    category: business
    files:
      - path: logs/business.log
        format: "{{time}} [{{level}}] [{{class}}.{{method}}] {{message}}"
        append: true
        maxSize: 10MB
        maxFiles: 5
```

5.2 Placeholders Disponibles

Le système de templates supporte plusieurs placeholders :

Placeholder	Description
{{time}}	Timestamp actuel
{{message}}	Message configuré dans la règle
{{value}}	Valeur de retour ou message d'exception
{{method}}	Nom de la méthode
{{class}}	Nom de la classe
{{args}}	Arguments de la méthode
{{exception}}	Détails de l'exception
{{level}}	Niveau de log (TRACE, DEBUG, INFO, WARN, ERROR)
{{category}}	Catégorie du logger (system, business)

5.3 Organisation Modulaire

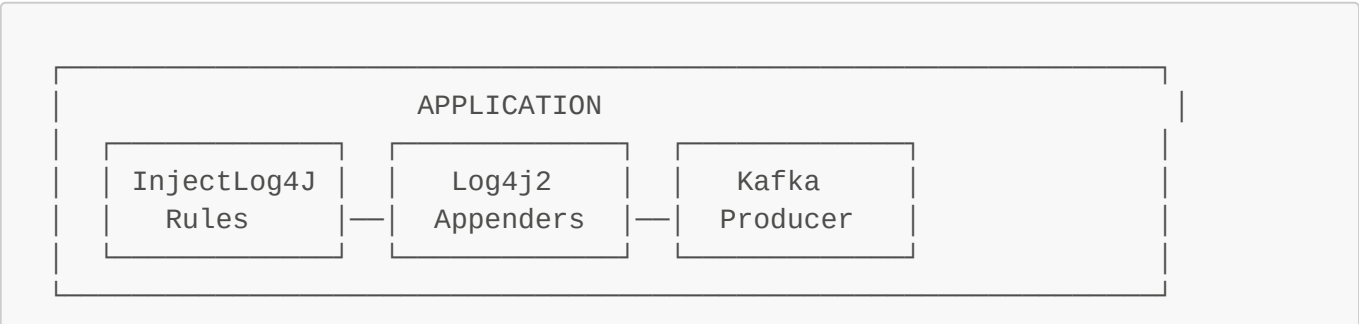
J'ai choisi d'organiser les règles par domaine fonctionnel. Cela facilite la maintenance et permet à chaque équipe de gérer ses propres règles :

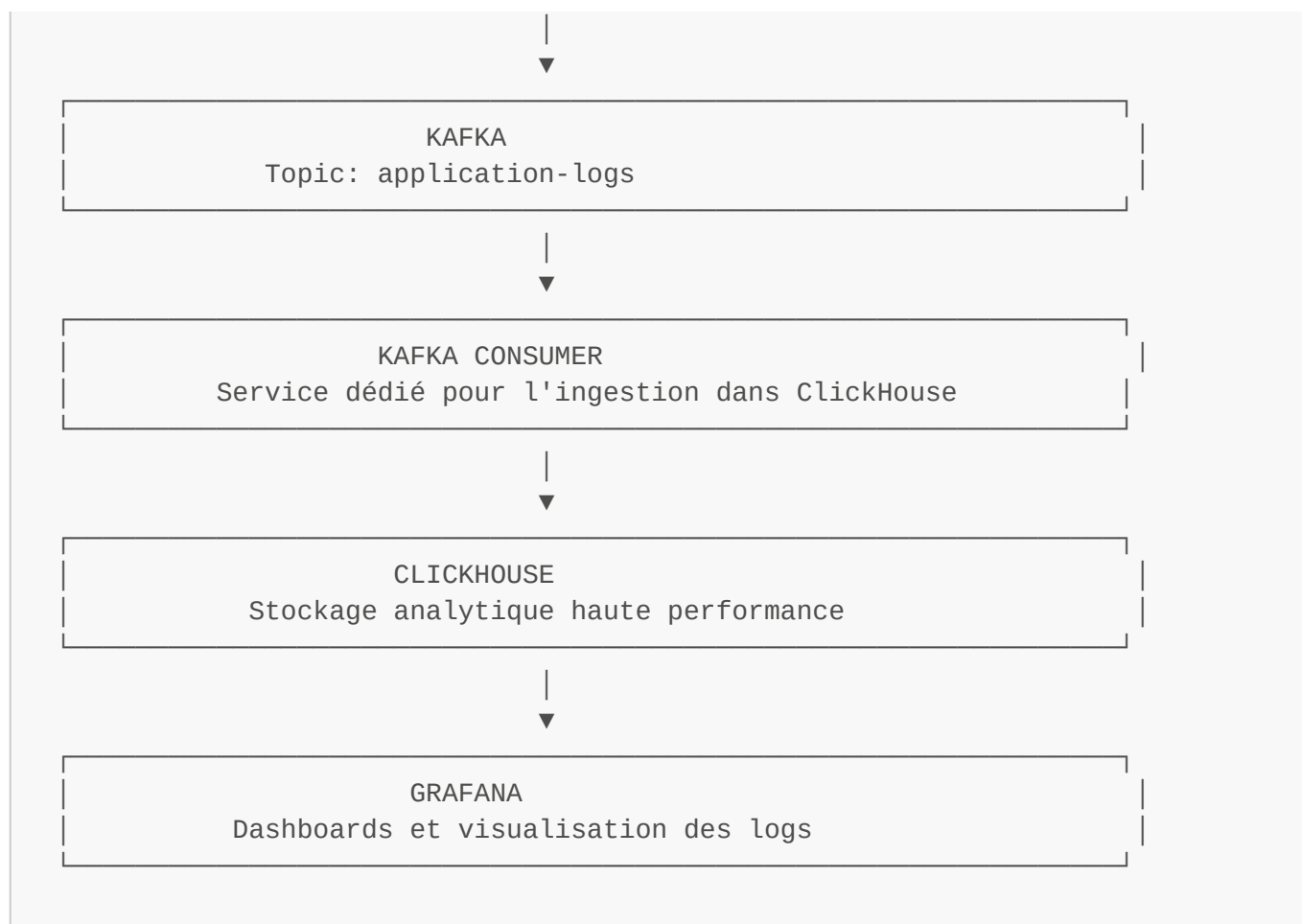
```
app/src/main/resources/  
├─ loggers.logging.rules.yaml      # Définition des loggers  
├─ user.logging.rules.yaml         # Règles pour UserService  
├─ product.logging.rules.yaml      # Règles pour ProductService  
├─ profile.logging.rules.yaml      # Règles pour ProfileService  
└─ system.logging.rules.yaml       # Règles infrastructure
```

6. Pipeline d'Observabilité

6.1 De l'Application à l'Analyse

Les logs ne sont qu'une partie de l'observabilité. J'ai intégré InjectLog4J dans un pipeline complet :





6.2 Configuration Log4j2

Le fichier `log4j2.xml` configure les différentes destinations :

```
<Configuration status="WARN" monitorInterval="30">
  <Properties>
    <Property name="LOG_PATTERN">
      %d{yyyy-MM-dd HH:mm:ss.SSS} [%t] %-5level %logger{36} - %msg%n
    </Property>
  </Properties>

  <Appenders>
    <!-- Console pour le développement -->
    <Console name="Console" target="SYSTEM_OUT">
      <PatternLayout pattern="${LOG_PATTERN}" />
    </Console>

    <!-- Fichiers avec rotation -->
    <RollingFile name="File" fileName="./logs/application.log"
      filePattern="./logs/application-%d{yyyy-MM-dd}-%i.log">
      <PatternLayout pattern="${LOG_PATTERN}" />
      <Policies>
        <TimeBasedTriggeringPolicy interval="1" />
        <SizeBasedTriggeringPolicy size="10MB" />
      </Policies>
    </RollingFile>
  </Appenders>
</Configuration>
```

```
<!-- Kafka pour l'agrégation centralisée -->
<Kafka name="Kafka" topic="application-logs">
  <PatternLayout pattern="${LOG_PATTERN}" />
  <Property name="bootstrap.servers">kafka:9092</Property>
</Kafka>
</Appenders>

<Loggers>
  <Logger name="fr.umontpellier.observability" level="info"
additivity="false">
    <AppenderRef ref="Console" />
    <AppenderRef ref="File" />
    <AppenderRef ref="Kafka" />
  </Logger>
</Loggers>
</Configuration>
```

6.3 Schéma ClickHouse

Les logs sont stockés dans ClickHouse pour une analyse rapide :

```
CREATE TABLE IF NOT EXISTS default.application_logs (
  timestamp DateTime DEFAULT now(),
  level String,
  logger String,
  message String,
  thread String
) ENGINE = MergeTree()
ORDER BY timestamp;
```

ClickHouse est particulièrement adapté aux logs car il excelle dans les requêtes analytiques sur de gros volumes de données.

6.4 Intégration avec OpenTelemetry

Le projet inclut également un traçage distribué avec OpenTelemetry. Cela permet de corréler les logs avec les traces :

- **Frontend Angular** : Instrumentation automatique des requêtes HTTP
- **Backend Spring Boot** : Agent OpenTelemetry Java
- **Collector** : Agrégation et export vers Jaeger

Cette intégration permet de naviguer d'une trace vers les logs associés, facilitant le debugging en production.

7. Application de Démonstration

Pour tester et démontrer InjectLog4J, j'ai développé une application complète de gestion d'utilisateurs et de produits. Elle n'est pas le cœur du projet, mais elle permet de voir le système de logging en action dans un contexte réaliste.

7.1 Stack Technique

- **Backend** : Spring Boot 3.4.1, Java 21, MongoDB
- **Frontend** : Angular 21 (avec traçage OpenTelemetry)
- **Infrastructure** : Docker Compose avec Kafka, ClickHouse, Grafana, Jaeger

7.2 Démarrage Rapide

```
# Installation des dépendances frontend
cd frontend && npm install && cd ..

# Lancement de toute l'infrastructure
cd docker && docker compose up --build
```

Services disponibles :

Service	URL	Description
Application	http://localhost:4200	Interface Angular
API Backend	http://localhost:8080/api	API REST
Grafana	http://localhost:3000	Dashboards (admin/admin)
Jaeger	http://localhost:16686	Traçage distribué

7.3 Voir les Logs en Action

Une fois l'application démarrée, on peut créer un utilisateur via l'API :

```
curl -X POST http://localhost:8080/api/users \
-H "Content-Type: application/json" \
-d '{"name":"Test User","email":"test@example.com","password":"secret"}'
```

Et observer les logs dans la console de l'application :

```
2025-12-27 14:30:00.123 [BUSINESS] [UserService.createUser] Creating new
user - result: User(id=123, name=Test User, ...)
```

8. Politique d'Utilisation de l'IA

8.1 Transparence sur l'Usage de l'IA

Dans un souci de transparence académique, je tiens à préciser le rôle qu'a joué l'intelligence artificielle dans ce projet.

8.2 Développement du Frontend et du Backend

Les applications **frontend Angular** et **backend Spring Boot** ont été développées avec l'assistance d'outils d'IA (GitHub Copilot, Claude). Ces outils m'ont permis de :

- **Accélérer le prototypage** : Génération rapide de boilerplate code pour les contrôleurs REST, les services et les composants Angular
- **Explorer les APIs** : Découverte des bonnes pratiques pour OpenTelemetry, Spring Boot Actuator et les configurations Docker
- **Débugger plus efficacement** : Identification rapide des erreurs de configuration et des problèmes d'intégration

Cependant, chaque morceau de code généré a été **relu, compris et adapté** à mes besoins spécifiques. L'IA a été un outil d'assistance, pas un substitut à la réflexion.

8.3 Refactoring et Principes SOLID

L'IA a été particulièrement utile pour les phases de **refactoring**. Une fois le code fonctionnel écrit, j'ai utilisé l'IA pour :

Vérifier le respect des principes SOLID :

Principe	Application avec l'aide de l'IA
Single Responsibility	L'IA m'a aidé à identifier les classes qui faisaient "trop de choses" et à les découper en composants plus cohérents
Open/Closed	Suggestions pour utiliser des interfaces et des abstractions plutôt que des implémentations concrètes
Liskov Substitution	Vérification que les sous-classes respectent les contrats de leurs classes parentes
Interface Segregation	Découpage d'interfaces trop larges en interfaces plus spécifiques
Dependency Inversion	Passage à l'injection de dépendances via constructeurs plutôt que l'instanciation directe

Exemples concrets de refactoring :

- Le **LogInjector** a été refactorisé pour utiliser le pattern **Factory** (**LogOutputFactory**) au lieu de créer directement les outputs
- Les sorties de logs ont été abstraites derrière l'interface **LogOutput**, permettant d'ajouter facilement de nouvelles destinations
- Le **SpoonProjectProcessor** a été simplifié en extrayant la logique de création du processeur dans une méthode protégée, facilitant les tests

8.4 Ce que l'IA n'a pas Fait

Il est important de noter que :

- **La conception globale** du système InjectLog4J (l'idée d'utiliser des fichiers YAML pour configurer le logging déclarativement) est le fruit de ma réflexion personnelle
- **Les choix architecturaux** (Spoon pour la transformation, le pattern Composite pour les sorties multiples) ont été décidés après recherche et comparaison d'alternatives
- **L'intégration dans le pipeline d'observabilité** (Kafka → ClickHouse → Grafana) a nécessité une compréhension approfondie de chaque outil
- **Le debugging et les ajustements** pour faire fonctionner l'ensemble ont été réalisés manuellement

8.5 Position sur l'Usage de l'IA

Je considère l'IA comme un **outil de productivité**, comparable à un IDE avancé ou à Stack Overflow. Elle permet de gagner du temps sur les tâches répétitives et d'explorer rapidement des solutions, mais elle ne remplace pas :

- La compréhension des concepts fondamentaux
- La capacité à prendre des décisions architecturales
- L'esprit critique pour évaluer la qualité du code généré
- La responsabilité de maintenir et faire évoluer le projet

Dans un contexte professionnel comme académique, je pense qu'il est essentiel d'être **transparent** sur l'utilisation de ces outils et de s'assurer qu'on maîtrise le code qu'on livre, qu'il ait été écrit manuellement ou avec assistance.

9. Conclusion et Perspectives

9.1 Limites Actuelles

Le projet a quelques limitations que j'aimerais adresser :

- **Transformation au runtime** : Actuellement, la transformation Spoon se fait au build. Une injection au runtime via instrumentation bytecode serait plus flexible
- **Performance** : L'évaluation des règles pourrait être optimisée avec un système de cache
- **Corrélation** : Intégrer automatiquement les trace IDs OpenTelemetry dans les logs

9.2 Évolutions Envisagées

Pour la suite, j'envisage plusieurs améliorations :

- **Plugin Maven/Gradle** : Simplifier l'intégration dans les projets existants
- **UI de configuration** : Une interface web pour définir les règles visuellement
- **Sampling intelligent** : Réduire le volume de logs en production sans perdre les informations importantes
- **Intégration IDE** : Plugin IntelliJ pour visualiser les règles appliquées à chaque méthode

10. Références

Documentation Technique

- **Spoon** : <https://spoon.gforge.inria.fr/>
- **Log4j2** : <https://logging.apache.org/log4j/2.x/>
- **Spring Boot** : <https://docs.spring.io/spring-boot/>

Observabilité

- **OpenTelemetry** : <https://opentelemetry.io/docs/>
- **Jaeger** : <https://www.jaegertracing.io/docs/>
- **Grafana** : <https://grafana.com/docs/>

Stockage

- **ClickHouse** : <https://clickhouse.com/docs/>
- **Apache Kafka** : <https://kafka.apache.org/documentation/>

Projet disponible sur : fr.umontpellier.fr/observability

Contact : Jérémy Hurel - Université de Montpellier