Configuration Injection Analysis in the E4L Platform

Alberto Finardi
University of Luxembourg
Luxembourg
alberto.finardi.001@student.uni.lu

Tommaso Crippa
University of Luxembourg
Luxembourg
tommaso.crippa.001@student.uni.lu

Jan Esquivel Marxen
University of Luxembourg
Luxembourg
jan.esquivel.001@student.uni.lu

Abstract

Configuration injection at different lifecycle stages is a fundamental practice in modern software engineering that enables flexibility, maintainability, and environment-specific customization without modifying source code. This report analyzes configuration injection opportunities in the Energy4Life (E4L) platform across three distinct phases: build-time, deployment-time, and startup/runtime. We examine existing hardcoded values and infrastructure configurations, evaluate their potential for externalization, and assess the feasibility and implications of making these elements configurable. Our analysis reveals multiple opportunities for improvement, ranging from simple property-based configurations to more complex infrastructure-level parameterization.

Note: Portions of this report were developed with the assistance of artificial intelligence tools to support idea generation, text refinement, and readability improvements. All content was reviewed and validated by the authors.

Keywords: Configuration Management, Software Engineering, Deployment, DevOps, Externalized Configuration

ACM Reference Format:
Alberto Finardi, Tommaso Crippa, and Jan Esquivel Marxen. 2025. Configuration Injection Analysis in the E4L Platform. In *Proceedings of Software Environment Engineering Course (SEE 25).* ACM, New York, NY, USA, 4 pages.

1 Introduction

Modern software systems must operate across diverse environments—development, staging, production—each with distinct requirements for database connections, external service endpoints, resource constraints, and operational policies. Hardcoding configuration values into source code or build artifacts creates rigidity, requiring recompilation or redeployment for simple environmental adjustments. Configuration injection, the practice of externalizing such values and injecting them at appropriate lifecycle stages, is a well-established pattern that improves software flexibility, security, and maintainability.

The Energy4Life (E4L) platform, a Spring Boot-based application for measuring carbon footprints, presents multiple opportunities for configuration injection. This analysis examines both existing infrastructure configurations and planned features:

Existing Infrastructure (Basic):

- Database credentials and connection strings via environment variables
- Build artifacts and version metadata via build-time configuration

Planned Features (Advanced):

- Quiz versioning A/B testing for experimentation
- UI theming for customization and white-labeling

For each example, we provide code evidence, analyze configuration injection at different lifecycle stages (build-time, deployment-time, startup/runtime), evaluate pros and cons, and assess implementation feasibility.

2 Example 1: Database Configuration (Deployment-Time)

2.1 Context

Database credentials and connection strings are environment-specific values that must differ between development, staging, and production. The E4L platform already uses deployment-time configuration for these values.

2.2 Evidence

File: `docker/docker-compose.backend.pre-prod.yml`

```
environment:
  - SPRING_DATASOURCE_URL=jdbc:mysql://
      e4l-db/e4lpreprod?...
  - SPRING_DATASOURCE_USERNAME=root
  - SPRING_DATASOURCE_PASSWORD=12345678
  - RESOURCES_STATIC_URL=http://localhost:8084/
  - TZ=Europe/Luxembourg
```

File: `src/main/resources/application.properties`

```
spring.datasource.url=jdbc:mysql://localhost:3306/
    e4l?serverTimezone=Europe/Paris
spring.datasource.username=root
spring.datasource.password=12345678
```

Spring Boot automatically reads environment variables and overrides `application.properties` values, enabling deployment-time configuration without code changes.

2.3 Analysis

Table 1. Database Configuration Analysis

Pros	Cons
Clean separation of code and configuration	Risk of secret exposure if not managed properly
Easy environment switching without rebuilding	Configuration sprawl across multiple files
Secrets managed externally from source control	Silent failures from typos in variable names
No code changes required	Requires coordination between dev and ops

Feasibility: Easy. This pattern is already implemented and working. Improvements could include using `.env` files or secret managers (Docker Secrets, HashiCorp Vault) for enhanced security, which would be moderate complexity.

3 Example 2: Build Artifact Naming (Build-Time)

3.1 Context

Build-time configuration involves parameters embedded into compiled artifacts. Currently, artifact names and versions are hardcoded in the build configuration.

3.2 Evidence

File: `lu.uni.e4l.platform.api.dev/build.gradle`

```
group 'lu.uni.e4l.platform'
version '1.0-SNAPSHOT'

bootJar {
    archiveFileName = 'e4l-server.jar'
    destinationDirectory = file("build/libs")
}
```

File: `lu.uni.e4l.platform.api.dev/Dockerfile`

```
FROM openjdk:20-jdk-slim
COPY build/libs/e4l-server.jar app.jar
ENTRYPOINT ["java", "-jar", "/app.jar"]
```

The artifact name `e4l-server.jar` is hardcoded, making versioned artifact management difficult.

3.3 Proposed Configuration

Make artifact naming configurable via Gradle properties:

```
// gradle.properties
artifactBaseName=e4l-server
version=1.0.0
```

```
// build.gradle
bootJar {
   archiveFileName = "${artifactBaseName}-${version}.jar"
}
```

This enables CI/CD pipelines to inject version numbers at build time: `./gradlew build -Pversion=1.2.3`

3.4 Analysis

Table 2. Build Artifact Configuration Analysis

Pros	Cons
Reproducible builds with versioned artifacts	Configuration changes require complete rebuild
Embed build metadata for traceability	Less flexible post-build
CI/CD friendly for automated versioning	Dockerfile must reference dynamic artifact name

Feasibility: Easy. Gradle provides robust support for project properties. Requires minimal changes to `build.gradle` and `Dockerfile`. No code changes needed, only build configuration updates.

4 Example 3: Feature Flags (Startup/Runtime)

4.1 Context

The project requires feature flags to enable/disable functionality (e.g., quiz versioning A/B testing, theming) for safe rollout and experimentation. This is a planned feature from the todo list.

4.2 Evidence

Feature flags are not currently implemented. Example hardcoded value that would benefit:

File: `src/main/java/lu/uni/e4l/platform/service/QuestionnaireService.java:25`

```
private static final String DEFAULT_QUESTIONNAIRE
    = "energy4life";
```

This could be made configurable to support A/B testing of quiz versions.

4.3 Proposed Configuration

Startup Injection (Simple):

Add to `application.properties`:

```
features.quiz-versioning.enabled=true
features.theming.enabled=false
questionnaire.default.name=energy4life
```

Implement in code:

```
@Service
public class QuestionnaireService {
    @Value("${questionnaire.default.name}")
    private String defaultQuestionnaire;

   @Value("${features.quiz-versioning.enabled}")
    private boolean quizVersioningEnabled;
}
```

Deployment-Time Override:

Via Docker Compose environment variables:

```
environment:
  - FEATURES_QUIZ_VERSIONING_ENABLED=true
  - QUESTIONNAIRE_DEFAULT_NAME=energy4life_v2
```

4.4 Analysis

Table 3. Feature Flags Configuration Analysis

Pros	Cons
Allows safe rollout and rollback	Requires code changes to add flag checks
Reduces deployment risk	Increases code complexity
Enables A/B testing without redeployment	Must coordinate dev and ops teams

Feasibility: Moderate. Requires code changes to replace hardcoded constants with `@Value` injections and add conditional logic. Simple property-based flags are straightforward, but comprehensive feature flag infrastructure requires more design and testing effort.

5 Example 4: UI Theming (Build/Runtime)

5.1 Context

The project aims to support UI theme customization (e.g., light/dark mode, branding colors) for improved user experience and potential white-label deployments. This is a planned feature from the todo list.

5.2 Evidence

Frontend Evidence:

- CSS/SCSS files in `lu.uni.e4l.platform.frontend.dev/src/css/` and `src/scss/`
- Static resources referenced: `resources.static.url` in `application.properties`
- Theme selection logic would be in `src/js/` components

Currently, CSS styles are likely hardcoded and not externally configurable.

5.3 Proposed Configuration

Build-Time Injection:

Generate theme-specific CSS bundles during build:

```
// gradle.properties
theme=light

// Build different themes:
./gradlew build -Ptheme=dark
```

Package theme-specific CSS files based on build parameter.

Runtime/Startup Injection:

Add to `application.properties`:

```
ui.theme.name=light
ui.theme.primary-color=#007bff
ui.theme.logo-url=/static/logo-light.png
```

Backend exposes theme config via API endpoint:

```
@RestController
public class ThemeController {
    @Value("${ui.theme.name}")
    private String themeName;

    @GetMapping("/api/theme")
    public ThemeConfig getTheme() { ... }
}
```

Frontend loads theme at startup and applies CSS variables.

Deployment-Time Injection:

Override per environment via Docker Compose:

```
environment:
  - UI_THEME_NAME=corporate
  - UI_THEME_PRIMARY_COLOR=#ff6600
```

5.4 Analysis

Table 4. UI Theming Configuration Analysis

Pros	Cons
Improves UX and accessibility	Requires frontend refactoring
Enables white-label deployments	Must ensure consistent CSS variable usage
Build-time option offers performance	Dynamic CSS loading adds complexity
Deployment-time allows per-env branding	Frontend/backend config contract needed

Feasibility: Moderate to Hard. Build time theming is easier (moderate complexity) but less flexible. Runtime theming requires significant frontend refactoring to support dynamic CSS variable loading and may need CSS architecture changes, making it harder. Backend API changes are straightforward.

6 Comparative Analysis

Table 5 summarizes the four configuration injection examples analyzed in this report.

Table 5. Configuration Injection Examples Comparison

Example	Injection Time	Code Changes	Key Benefit	Feasibility
Database Config	Deployment	None	Environment separation	Easy
Build Artifacts	Build	None (config only)	Versioned artifacts	Easy
Feature Flags/Quiz A/B	Startup/Runtime	Moderate	A/B testing	Moderate
UI Theming	Build/Runtime	Significant	White-labeling	Mod-Hard

6.1 Configuration Injection Timing Analysis

The examples demonstrate configuration injection at different lifecycle stages:

Build-Time (Example 2, Example 4):

- Artifact naming and versioning embedded during compilation
- Theme-specific CSS bundles for performance
- Pros: Reproducible, optimized artifacts
- Cons: Requires rebuild for changes

Deployment-Time (Example 1, all examples):

- Database credentials via Docker Compose environment variables
- All examples support environment-specific overrides
- Pros: Clean code/config separation, no rebuild needed
- Cons: Configuration, potential secret exposure

Startup/Runtime (Example 3, Example 4):

- Feature flags and quiz configuration via `application.properties`
- Theme settings loaded at application startup
- Pros: Maximum flexibility, no redeployment needed
- Cons: Requires code changes to support, runtime-only error discovery

6.2 Implementation Complexity

Basic Examples (Easy): Database configuration and build artifacts require minimal or no code changes. These leverage existing Spring Boot and Gradle capabilities.

Advanced Examples (Moderate-Hard): Feature flags and theming require code refactoring, new service layers, and careful testing. Theming additionally requires frontend architecture changes.

7 Conclusion

This analysis examined four configuration injection examples in the E4L platform, spanning basic infrastructure configurations and advanced planned features across different lifecycle stages (build time, deployment time, and startup/runtime).

The examples demonstrate a clear progression in implementation complexity:

Basic Examples (Easy):

- **Database Configuration** already works via environment variables with no code changes required
- **Build Artifacts** need only Gradle configuration updates for versioned builds

Advanced Examples (Moderate-Hard):

- **Feature Flags/Quiz A/B** require moderate code refactoring to replace hardcoded values with property injection
- **UI Theming** requires significant frontend and backend changes, with build-time being easier than runtime implementation

The analysis reveals that configuration injection provides immediate value for basic infrastructure (environment separation, artifact management) with minimal effort, while advanced features (A/B testing, theming) require more investment but enable experimentation and customization capabilities.

Spring Boot's native support for externalized configuration significantly reduces technical complexity for the basic examples. The advanced examples demonstrate that even complex features can be made configurable, though careful planning and testing are required. The choice of injection timing (build, deployment, or runtime) depends on the balance between flexibility and implementation complexity.

By adopting these configuration injection practices, the E4L platform achieves separation of concerns, environment-specific customization, and operational agility - all fundamental principles of modern DevOps practices.