Enhancing Logging with @Log and @Slf4j in Spring Boot Applications

Alexander Obregon Follow 8 min read · Sep 21, 2023

295 5

Image Source

Spring Boot has become a popular choice for developing enterprise-grade applications due to its ease of use, powerful features, and strong ecosystem.

One aspect that often gets overlooked but is essential for any application,

with the assistance of the Lombok library, developers can further simplify their logging approach using annotations like `@Log` and `@Slf4j`. This post will explore these annotations and how to effectively use them in Spring Boot applications.

I publish free articles like this daily, if you want to support my work and get access to exclusive content and weekly recaps, consider subscribing to my Substack.

The Basics of Logging in Spring Boot

When working with Spring Boot, developers are provided with a powerful and configurable logging system out-of-the-box. This section provides an overview of the essential elements and foundational knowledge of this system.

Spring Boot's Default Logging Framework

Spring Boot, by default, includes the Simple Logging Facade for Java (SLF4J) coupled with Logback. SLF4J acts as an abstraction layer, meaning you can plug in your preferred logging framework at deployment time, while Logback serves as the default implementation. This combination gives developers a flexible and highly configurable logging setup.

Understanding Logging Levels

In the world of logging, not all messages are created equal. Messages are categorized by severity or importance, known as logging levels. Spring Boot supports the standard levels, which are:

- **ERROR:** Denotes that something failed, and the application might not be able to continue running.
- **WARN:** Indicates a potential problem that might not immediately affect functionality but warrants attention.
- **INFO:** Provides general information about the application's operation. Typically used to confirm things are working as expected.
- **DEBUG:** Offers detailed insights for developers to diagnose issues or understand the flow.
- **TRACE:** Gives more granular details than DEBUG, often including iterative or repetitive processes.

Each level is inclusive of the levels above it. For instance, if you set the level to WARN, you'll also see ERROR messages, but not INFO, DEBUG, or TRACE.

Configuring Logging in `application.properties`

Spring Boot allows developers to configure the logging system using the `application.properties` (or `application.yml`) file. Here are some common configurations:

- **Setting Global Logging Level:** To set a base level for all loggers:

```
logging.level.root=WARN
```

- **Setting Specific Logging Level:** To define a specific level for a particular package or class:

```
logging.level.org.springframework.web=DEBUG
logging.level.com.myapp.service=INFO
```

- **Log File Output:** By default, logs are printed to the console. If you want them saved to a file:

```
logging.file.name=myapp.log
```

- **Log File Rotation:** For larger applications, logs can grow rapidly. To manage size, you need to configure log rotation directly in your logging framework's configuration file. Since Spring Boot uses Logback by default, you can set up log rotation in a `logback-spring.xml` file placed in your `src/main/resources` directory:

```
<configuration>
    <appender name="FILE" class="ch.qos.logback.core.rolling.RollingFileAppender
        <file>myapp.log</file>
        <rollingPolicy class="ch.qos.logback.core.rolling.SizeAndTimeBasedRollin
            <!-- daily rollover and up to 10MB per file -->
            <fileNamePattern>myapp.%d{yyyy-MM-dd}.%i.log</fileNamePattern>
            <maxFileSize>10MB</maxFileSize>
            <maxHistory>10</maxHistory>
        </rollingPolicy>
        <encoder>
            <pattern>%d{yyyy-MM-dd HH:mm:ss} - %logger{36} - %msg%n</pattern>
        </encoder>
    </appender>

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

Alternatively, you can specify a custom Logback configuration file in your `application.properties`:

```
logging.config=classpath:logback-spring.xml
```

Log Format Customization

Spring Boot's default log output is concise and developer-friendly. However, for specific requirements, you might want to customize the log pattern. Using `logging.pattern.console` and `logging.pattern.file`, you can define custom patterns for console and file outputs, respectively.

For instance:

```
logging.pattern.console=%d{yyyy-MM-dd HH:mm:ss} - %logger{36} - %msg%n
```

This pattern includes the timestamp, logger name (up to 36 characters), and the actual message.

Introduction to Lombok and Logging Annotations

Project Lombok is a boon for Java developers when it comes to reducing boilerplate code. While Java's verbosity can be an advantage for understanding program logic, it can become a liability, especially with mundane tasks like logging setup. Here's where Lombok steps in to streamline the process.

What is Lombok?

Lombok is a compile-time annotation processor. Instead of you writing repetitive code or relying on your IDE to generate it, Lombok provides annotations to instruct the compiler to generate the code on your behalf. This not only keeps your codebase cleaner but also makes the development process faster and less error-prone.

Logging Annotations by Lombok

While Lombok offers various annotations for diverse tasks, like `@Data` for getters, setters, and other common methods, we'll focus on the logging annotations:

- **@Slf4j:** This is the most commonly used logging annotation for Spring Boot applications. When applied to a class, it automatically creates a static SLF4J logger instance named `log`, targeting the SLF4J logging facade.

```
@Slf4j
public class MyService {
    public void someServiceMethod() {
        log.info("Service method called using @Slf4j");
    }
}
```

- **@Log:** This annotation is used for applications relying on the `java.util.logging` framework. Similar to `@Slf4j`, it provides a static logger instance named `log`.

```
@Log
public class LegacyService {
    public void legacyMethod() {
        log.info("Legacy method logged with @Log");
    }
}
```

Benefits of Using Lombok’s Logging Annotations

- **Consistency:** Using annotations creates a uniform logging setup across the application, providing a standardized method to add logging to any class.
- **Reduced Boilerplate:** Manual logger instantiation is no longer required for every class, reducing the number of lines of code and improving maintainability.
- **Focus on Business Logic:** By removing repetitive tasks, developers can dedicate more time to implementing business logic, leading to higher-quality code and faster development.
- **Refactoring Ease:** When class names are updated, there is no need to modify logger declarations manually, as Lombok automatically updates them during compilation.

Integrating Lombok with Spring Boot

To use Lombok with Spring Boot, simply add the Lombok dependency to your project’s build file:

For Maven:

```
<dependency>
    <groupId>org.projectlombok</groupId>
    <artifactId>lombok</artifactId>
    <version>CheckLatestVersion</version>
    <scope>provided</scope>
</dependency>
```

For Gradle:

```
compileOnly 'org.projectlombok:lombok:CheckLatestVersion'
annotationProcessor 'org.projectlombok:lombok:CheckLatestVersion'
```

Note: Check for the latest version of Lombok before adding the dependency.

Using @Log vs. @Slf4j

When diving into the logging landscape, especially within the Spring Boot ecosystem, two Lombok annotations often come to the forefront: `@Log` and `@Slf4j` . While both annotations facilitate logging by eliminating boilerplate, understanding the distinctions between them can assist developers in choosing the right fit for their applications.

Origins and Frameworks

@Slf4j:

- **Origin:** The acronym SLF4J stands for Simple Logging Facade for Java.
- **Framework Target:** This annotation is crafted specifically for the SLF4J logging facade. Given that SLF4J provides an abstraction for various logging frameworks, using @Slf4j allows flexibility. If, down the road, you decide to switch from Logback (Spring Boot's default) to another framework like Log4j2, SLF4J makes the transition seamless.

@Log:

- **Origin:** This annotation takes its name directly from the `java.util.logging` package, often abbreviated as JUL.
- **Framework Target:** `@Log` is tailored for the `java.util.logging` framework, which is Java's built-in logging mechanism. While not as popular or versatile as SLF4J in the Spring ecosystem, JUL has its place in legacy or specific Java applications where using the built-in logging mechanism is advantageous.

Usage Scenarios

@Slf4j:

- **Modern Spring Boot Applications:** Given that Spring Boot defaults to SLF4J with Logback, `@Slf4j` is the annotation of choice for most Spring Boot projects.

- **Interoperability Needs:** If there is a potential need to switch between logging frameworks, SLF4J's facade mechanism backed by @Slf4j supports seamless transitions.

@Log:

- **Legacy Applications:** In legacy Java applications where `java.util.logging` is deeply entrenched, introducing @Log simplifies logging in legacy applications that use `java.util.logging`.
- **Built-in Java Environments:** For projects where minimizing external dependencies is crucial, leaning on Java's built-in logging mechanism with the assistance of `@Log` might be beneficial.

How to Implement

For both annotations, implementation is straightforward. After making sure Lombok is integrated with your project, simply annotate the class:

- For `@Slf4j`:

```
@Slf4j
public class OrderService {
    public void placeOrder() {
        log.info("Order placed successfully using SLF4J");
    }
}
```

- For `@Log`:

```
@Log
public class InventoryChecker {
    public void checkStock() {
        log.info("Stock checked using java.util.logging");
    }
}
```

In both cases, Lombok creates a static logger instance named `log` for you.

Best Practices with @Slf4j and Logging

Logging effectively is as much about technique as it is about the tools. While `@Slf4j` eliminates boilerplate and simplifies logger instantiation, it's vital to understand and adhere to logging best practices to make the most of it.

Log Meaningful Messages

Make sure that each log message provides context and is clear enough for someone unfamiliar with the code to understand. Ambiguous messages like "Error occurred" should be avoided.

```
@Slf4j
public class PaymentService {
    public void processPayment(Payment payment) {
        if (payment == null) {
            log.error("Payment processing failed due to null payment object.");
        }
        // ...
    }
}
```

Use Appropriate Logging Levels

Misusing log levels can lead to overlooked important information or excessive logging. Use the appropriate level:

- **ERROR:** For serious issues that may prevent the application from continuing.
- **WARN:** For potential problems that don't halt operation.
- **INFO:** General operational messages about the application state.
- **DEBUG:** Messages useful for debugging, but too verbose for general logs.
- **TRACE:** Very detailed messages, typically used for intricate debugging.

Avoid Logging Sensitive Information

Never log sensitive information like passwords, credit card numbers, or personally identifiable information (PII). This is a security best practice and, in many jurisdictions, a legal requirement.

Use Parameterized Logging

Instead of string concatenation, utilize parameterized logging provided by SLF4J. This approach is efficient and can prevent unnecessary string creation.

```
String orderId = "O12345";
log.info("Processing order with ID: {}", orderId);
```

Handle Exceptions Properly

When logging exceptions, it's crucial to log the entire stack trace to diagnose the root cause. SLF4J makes this easy:

```
try {
    // some code that can throw an exception
} catch (Exception ex) {
    log.error("An error occurred while processing", ex);
}
```

Don’t Rely Solely on Logs for Monitoring

While logs are invaluable for diagnostics, they shouldn’t be the only monitoring tool. Metrics, alerts, and other monitoring tools should be used in tandem with logs.

Rotate and Archive Logs

Set up your logging system to rotate logs, preventing any single file from becoming too large and archiving older log files for future analysis. This can be managed by configuring a `logback-spring.xml` file as described earlier. Spring Boot itself does not directly support `logging.file.max-size` and `logging.file.max-history` properties.

Avoid Logging Inside Tight Loops

Logging inside loops, especially tight loops, can slow down an application significantly and generate enormous log files. Avoid logging inside loops unless absolutely necessary. If logging within a loop is unavoidable, make sure it is conditional and limited to higher logging levels, such as `INFO` or `WARN`.

Stay Consistent

Maintain consistency in logging patterns across your application. It improves readability and allows automated tools to parse logs efficiently.

Regularly Review and Prune Logs

Logs can often contain “log noise” — messages that were once useful but now clutter the logs. Regularly review and prune such messages, making sure logs remain a valuable resource.

Conclusion

Logging is a vital part of any application. While Spring Boot offers a comprehensive logging system out-of-the-box, using annotations like @Log and @Slf4j from Lombok reduces boilerplate code and provides a straightforward way to log messages. This allows developers to concentrate on building functional and efficient applications. As with all tools, using them properly and thoughtfully is key. Follow best practices, and your Spring Boot application logs will be a valuable resource.

1. *SLF4J Official Documentation*
2. *Project Lombok Documentation*
3. *Spring Boot Logging Guide*

Thanks for reading! If you found this helpful, highlighting, clapping, or leaving a comment really helps me out.

Spring Boot icon by Icons8

Spring Boot   Spring Framework   Java   Slf4j   Programming

Written by Alexander Obregon

26K followers · 15 following

Follow

I post daily about programming topics and share what I learn as I go. For recaps, exclusive content, and to support me: https://alexanderobregon.substack.com

Responses (5)

Write a response

What are your thoughts?

Mary Rose
Dec 5, 2024

Hi good morning everyone

18   Reply

Manojmanu
Jan 24, 2024

Hi

6 1 reply Reply

Salihu umar
Jan 23, 2024

Simple and concise

6 1 reply Reply

See all responses

More from Alexander Obregon

Alexander Obregon

Building Real-time Applications with Python and WebSockets

Real-time applications have become much more common over the past decade, enablin...

Jun 2, 2024 44 4

Alexander Obregon

Using Spring's @Retryable Annotation for Automatic Retries

Software systems are unpredictable, with challenges like network delays and third-...

Sep 16, 2023 400 8

Alexander Obregon

How Spring Boot Auto-Configuration Works

Introduction

Nov 18, 2024 114 3

Alexander Obregon

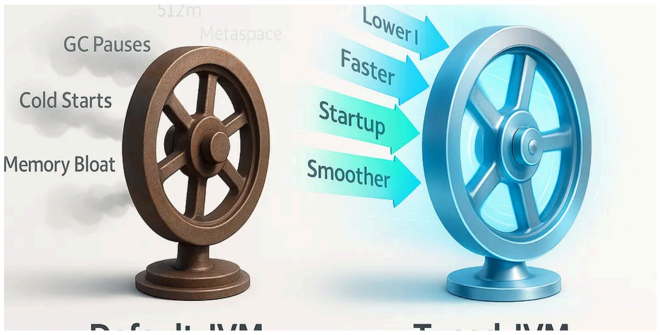
Building RESTful APIs with C++

Introduction

Jun 29, 2024 74 3

See all from Alexander Obregon

Recommended from Medium



Concurrent Mind

I Doubled My Spring Boot App's Speed With Just 5 JVM Flags

My Spring Boot app was embarrassingly slow.

3d ago 66

In Stackademic by Kavya's Programming Path

Advanced Spring Boot Concepts in Java (That Most Developers Still...

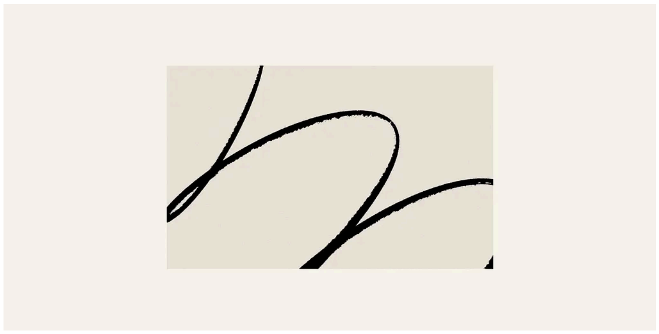
If you think you already know Spring Boot inside out, this article will challenge that beli...

Jul 18 220 4

In Javarevisited by Mohit Bajaj

How I Optimized a Spring Boot Application to Handle 1M...

Discover the exact techniques I used to scale a Spring Boot application from handling 50K...



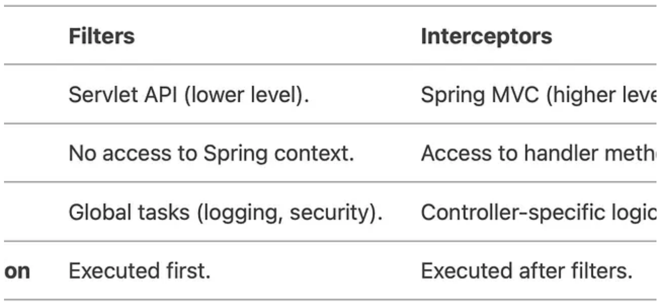
Ujjawal Rohra

5 Spring Boot Patterns That Separate Senior Developers From...

Let me ask you something honest

Mar 2 2.8K 76

Jul 5 180 3



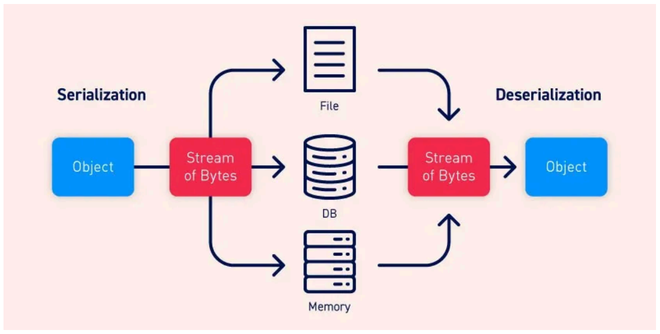
	Filters	Interceptors
	Servlet API (lower level).	Spring MVC (higher leve
	No access to Spring context.	Access to handler meth
	Global tasks (logging, security).	Controller-specific logic
on	Executed first.	Executed after filters.

Bolot Kasybekov

Mastering Spring Boot Filters and Interceptors: A Guide to Request...

Introduction In Spring Boot applications, handling HTTP requests often requires mor...

Mar 9 10



The Latency Gambler

How I Made My Java Service 70x Faster by Rethinking JSON...

The Problem You Don't Notice—Until You Scale

Jul 16 656 17

See more recommendations