Class-28: Application Logging

by Pial Kanti Samadder

Why Logging is Essential

Purpose of Logging

- Provides visibility into application behavior at runtime
- Helps diagnose issues without attaching a debugger
- Supports auditing, compliance, and traceability of system activities

Key Benefits

- **Debugging:** Quickly identify root causes of errors and failures
- **Monitoring:** Track system performance and health in real-time
- **Security & Auditing:** Retain evidence of important actions and access attempts
- **Operational Insight:** Understand usage patterns and optimize performance

How Spring Boot Integrates Logging Frameworks

Spring Boot integrates logging frameworks like **Logback** and **Log4j2** by acting as an intermediary that configures and orchestrates these systems seamlessly during application startup.

This integration is built on a well-defined set of defaults and conventions, which can be extended or replaced by user configurations.

Dependency Resolution and Initialization

The choice of logging framework in Spring Boot starts with dependency management:

- **Default Logging Setup**: The `spring-boot-starter-logging` dependency is included by default in all Spring Boot applications. It pulls in SLF4J (Simple Logging Facade for Java) as the logging API and Logback as the default implementation.
- **Switching to Log4j2**: If a developer prefers Log4j2, they can include the `spring-boot-starter-log4j2` dependency. Spring Boot will automatically exclude **Logback** and replace it with **Log4j2** without requiring any manual exclusions.
- **In Case of Conflict (Both Present):**
  Spring Boot applies its **dependency management rules** and **selects one logging implementation**.
  → If you explicitly specify one (e.g., Log4j2), that one wins.
  → If nothing is specified, **Logback remains the default.**

SLF4J - The Logging Facade

SLF4J (Simple Logging Facade for Java) is a **common logging API** that your application uses to write log statements.

It **does not** perform the actual logging — it delegates to an underlying logging framework like **Logback** or **Log4j2**.

Why Use SLF4J?

Benefit	Explanation
Framework Agnostic	Your code calls SLF4J — not **Logback** or **Log4j2** directly.
Easy to Switch Implementations	Change the logging engine by changing dependencies, **no code changes needed**.
Cleaner, Standard Logging Style	Provides a consistent logging pattern across your project.

How Spring Boot Handles It

- Your code logs through **SLF4J**.
- SLF4J **routes the log** to the **active logging engine**.
- The active engine (**Logback** by default) formats, filters, and writes the log.

Default Logging Configuration Process

Spring Boot configures logging automatically during application startup through the following steps:

1. **Detecting User Configuration**
   - Spring Boot first checks for custom logging config files (e.g., `logback-spring.xml`, `log4j2-spring.xml`, `.properties`, or `.yml`) in the **resources** directory.
   - If found, Spring Boot **uses these custom settings**.
2. **Applying Default Settings**
   If no custom configuration is provided:
   - Spring Boot applies **built-in default logging settings**.
   - **Root logger** level defaults to **INFO**.
   - Logs are written to the **console** using a standard formatting pattern.
3. **Enabling Framework-Specific Features**
   Once the logging framework is initialized, Spring Boot activates its unique features.

Compare Java Logging Frameworks

- **Log4j (1.x)**
  - Introduced hierarchical loggers and levels
  - Compatible with SLF4J
  - End-of-life 2015 → legacy projects only
- **Logback**
  - Successor to Log4j
  - Faster performance, native SLF4J support
  - Advanced filtering and automatic config reload
  - Actively maintained, widely used
- **Log4j2**
  - Modern, high-performance framework
  - SLF4J support, async logging, lambda evaluation
  - Garbage-free mode for low-latency applications
  - Recommended choice for new projects

Logback replaced **Log4j 1.x**; **Log4j2** improves on both with better performance and advanced features.

Logback and Log4j2 Initialization Differences

1. **Logback**
   - Spring Boot detects:
     - `logback-spring.xml`, `logback.xml`, `logback-spring.yml`
   - Uses an **internal appender chain**:
     - Examples: `ConsoleAppender`, `FileAppender`
     - Format example: `%d %level %logger{36} - %msg%n`
   - Supports **dynamic configuration updates at runtime**
   - **Hierarchical logger structure**
2. **Log4j2**
   - Spring Boot detects:
     - `log4j2-spring.xml`, `log4j2.xml`
   - Uses a **plugin-based architecture:**
     - Supports custom appenders, filters, and layouts
   - Fully supports **asynchronous logging for high-throughput apps**
   - **Hierarchical logger structure**, similar to **Logback**

Logging Levels & Severity in Spring Boot

What is a Logging Level?

- A **logging level** defines the **importance or verbosity of a log message**.
- Determines **which messages are recorded** based on their priority.
- Helps developers control **how much information is logged** during application runtime.

What is Severity?

- **Severity** indicates the **impact or seriousness of an event**.
- Higher-severity messages usually indicate **errors or critical issues**, while lower-severity messages are **informational or debug-related**.
- Logging levels are essentially a **hierarchical representation of severity**.

Standard Logging Levels (Low → High Severity)

Level	Description
TRACE	Very detailed diagnostic info (lowest severity)
DEBUG	Debugging info for development
INFO	General runtime info about application behavior
WARN	Potentially harmful situations requiring attention
ERROR	Severe errors causing application failure
OFF	Logging disabled (highest "severity")

Filtering Mechanism

- **Log messages below the configured level are ignored.**
- Example: Logger level = **INFO** → only **INFO, WARN, ERROR** are logged; **DEBUG, TRACE** are ignored.

Level Hierarchy:
TRACE < DEBUG < INFO < WARN < ERROR < OFF

Default Log Level in Spring Boot

Default Behavior

- Spring Boot uses **Logback** by default.
- The **root logger** is set to:

```
INFO
```

- Logs include **INFO, WARN, and ERROR** messages by default.
- Spring framework and other internal logs also inherit default levels:
  - `org.springframework` → INFO
  - `org.hibernate` → INFO

Changing Log Levels via `application.yml`

```
logging:
  level:
    root: ERROR        # Set root logger to ERROR
    com.example: DEBUG  # Set package-specific logger
    org.springframework: WARN
```

Explanation:

- `root` → default level for all loggers
- `com.example` → override for a specific package
- Messages **below the configured level are ignored**

Logging in Spring Boot

- ### Using SLF4J

```
@Service
public class MyService {
    private static final Logger logger = LoggerFactory.getLogger(MyService.class);

    public void process() {
        logger.debug("Debug message");
        logger.info("Info message");
        logger.warn("Warning message");
        logger.error("Error message");
    }
}
```

- ### Using Lombok (`@Slf4j`)

```
@Slf4j
@Service
public class MyService {

    public void process() {
        log.debug("Debug message");
        log.info("Info message");
        log.warn("Warning message");
        log.error("Error message");
    }
}
```

- Lombok's `@Slf4j` **automatically generates** the logger instance.
- Reduces **boilerplate code** and improves readability.
- Works seamlessly with Spring Boot and SLF4J.

Setting Log Pattern in Spring Boot

What is a Log Pattern?

- A **log pattern** defines the **format of each log message**.
- Common elements include:
  - Timestamp (`%d`)
  - Log level (`%level`)
  - Logger name (`%logger{36}`)
  - Thread name (`%thread`)
  - Message (`%msg`)

Configure Pattern in `application.yml`

```
logging:
  pattern:
    console: "%d{yyyy-MM-dd HH:mm:ss} [%thread] %-5level %logger{36} - %msg%n"
```

Example Output:

```
2025-11-07 21:10:00 [main] INFO  MyService - Processing started
2025-11-07 21:10:01 [main] WARN  MyService - Potential issue detected
```

- `logging.pattern.console` → pattern for **console logs**
- `logging.pattern.file` → pattern for **file logs** (if using `logging.file.name`)
- You can include **MDC variables** in the pattern:

```
logging:
  pattern:
    console: "%d [%thread] %-5level %logger{36} [userId=%X{userId}] - %msg%n"
```

Why Use MDC (*Mapped Diagnostic Context)*

Scenario:

We are building a **money transfer service**. The service logs messages during a transfer:

```
@Slf4j
@Service
public class MoneyTransferService extends TransferService {

    @Override
    protected void beforeTransfer(long amount) {
        log.info("Preparing to transfer " + amount + "$.");
    }

    @Override
    protected void afterTransfer(long amount, boolean outcome) {
        log.info("Has transfer of " + amount + "$ completed successfully? " + outcome + ".");
    }
}
```

Problem:

- Only the **amount** is accessible in log messages.
- Cannot log **transaction ID** or **sender**.
- Hard to trace logs when **multiple transfers run concurrently**.

Solution with MDC

- **Define a logging pattern with MDC placeholders**

```
<pattern>%d [%thread] %-5level %logger{36} - %msg - tx.id=%X{transaction.id} tx.owner=%X{transaction.owner}%n</pattern>
```

- **Set MDC variables before logging**

```
MDC.put("transaction.id", tx.getTransactionId());
MDC.put("transaction.owner", tx.getSender());
```

- **Log as usual**

```
log.info("Preparing to transfer " + tx.getAmount() + "$.");
log.info("Has transfer of " + tx.getAmount() + "$ completed successfully? " + outcome + ".");
```

- **Clear MDC after operation**

```
MDC.clear();
```

Example Output:

```
638 [pool-1-thread-2] INFO MoneyTransferService - Transfer completed - tx.id=2 tx.owner=Marc
666 [pool-1-thread-1] INFO MoneyTransferService - Transfer started   - tx.id=5 tx.owner=Susan
```

- MDC attaches **contextual info to the thread**.
- Makes logs **traceable and user-specific**.
- Essential for **multi-threaded applications**.

MDC and Thread Pools

- MDC stores contextual information using **ThreadLocal**, making it thread-safe.
- **Problem:** ThreadLocal + thread pools can cause **data leakage** between tasks.

How Issues Happen:

1. Thread is borrowed from a thread pool.
2. Contextual info is set in MDC (`MDC.put()`).
3. Logs use this info during execution.
4. MDC is **not cleared**.
5. Thread is returned to the pool.
6. Later, the same thread is reused for a **different task**, but MDC still holds the old context.
7. Logs may show **incorrect or mixed contextual data**.

Solutions:

- **Always clear MDC at the end of execution:**

```
MDC.clear();
```

- **Use ThreadPoolExecutor hooks** to automatically clear MDC after each task.
- Avoid manual errors; ensure **context cleanup** for reliable logging.

External Log Appenders in Spring Boot

What are External Log Appenders?

- Appenders define **where log messages are written**.
- Examples:
  - Console (default)
  - File
  - Database
  - Remote log servers (ELK, Graylog, Splunk)
  - Messaging systems (Kafka, JMS)
- Allows **flexible log storage** and integration with monitoring tools.

Using `application.yml` for simple file logging:

```
logging:
  file:
    name: logs/application.log
  pattern:
    console: "%d{yyyy-MM-dd HH:mm:ss} [%thread] %-5level %logger{36} - %msg%n"
    file: "%d{yyyy-MM-dd HH:mm:ss} [%thread] %-5level %logger{36} - %msg%n"
```

- Spring Boot automatically configures a **FileAppender** using these settings.
- No configuration file required for **basic file logging**.