Logging in Spring Boot

What is Logging?

Logging means recording important information about what your application is doing while it runs — like events, errors, or messages — to the **console or a file**.

Why do we need Logging?

We need logging to:

1. 🐞 **Find and fix bugs** — helps understand what went wrong.
2. 🧠 **Monitor application behavior** — see what your app is doing.
3. 📜 **Keep records** — store logs for audits or future reference.
4. ⚙ **Debug in production** — find issues without stopping the app.

---

1. Logging in Spring Boot

🔧 Implementation Steps:

Spring Boot uses **Spring Boot Starter Logging**, backed by **Logback** by default.

- **Basic Setup:**

In `application.properties`:

```
# Set log level (OFF, ERROR, WARN, INFO, DEBUG, TRACE, ALL)
logging.level.org.springframework=INFO logging.level.com.yourpackage=DE

# File Logging logging.file.name=app.log
logging.file.path=logs
```

- **Log Output Format (Optional):**

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

Spring Boot includes logging by default:

When you add any starter like:

```
<dependency>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-web</artifactId>
</dependency>
```

Spring Boot automatically includes:

```
<dependency>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-logging</artifactId>
</dependency>
```

Example: Logging in a Spring Boot Controller

```java
package com.vijay.demo.controller;

import org.slf4j.Logger;
import org.slf4j.LoggerFactory;
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.RestController;

@RestController
public class LoggingController {

    // ✅ Create Logger instance for this class
    private static final Logger logger = LoggerFactory.getLogger(LoggingController.class);

    @GetMapping("/log-demo")
    public String logDemo() {
        // ✅ Use logger inside a method
        logger.trace("This is a TRACE message");
        logger.debug("This is a DEBUG message");
        logger.info("This is an INFO message");
        logger.warn("This is a WARN message");
        logger.error("This is an ERROR message");

        return "Check console for log messages!";
    }
}
```

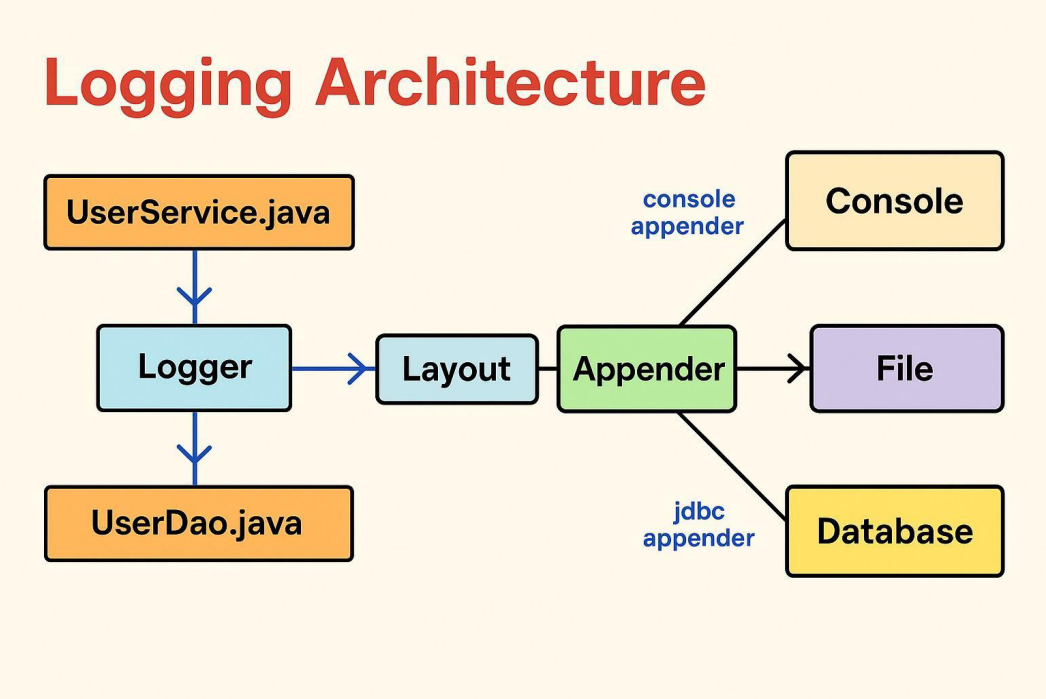
What is an Appender?

An **Appender** is a component in a logging framework (like **Logback** or **Log4j**) that **decides where your log messages go**.

In Simple Words

An **Appender** tells the logger **where to write the logs**, such as:

- 🖥️ Console
- 📁 File
- 📤 Database
- ☁️ Remote server



What's inside `spring-boot-starter-logging`?

It brings in:

- Logback (default backend)
- SLF4J (logging facade)

✅ What is the default logging in Spring Boot?

- **Default Logging Framework**: **Logback**
- **Logging Facade**: **SLF4J** (Simple Logging Facade for Java)
- **Logging Starter**: Included via `spring-boot-starter-logging` (auto-added)

📌 What is the default log level in Spring Boot?

- ✅ **Default Log Level**: `INFO`

- So by default, only logs at level `INFO`, `WARN`, and `ERROR` are shown.
  `DEBUG` and `TRACE` messages are **hidden** unless explicitly enabled.

📌 To enable `DEBUG` or `TRACE`:

- In `application.properties`:

```
logging.level.root=DEBUG
```

✅ When you run a Spring Boot app, where can you see the logs?

- By default, Spring Boot logs are visible in the **console (terminal/output window)**.

🗂️ To write logs to a file (optional):

- Add this in `application.properties`:
- `logging.file.name=app.log`

📁 Where is the file created?

➡ **Location:**

```
<your-project-root>/logs/app.log
```

- `logging.file.path=logs/` means a folder named `logs` will be created inside your **project directory**

- `logging.file.name=app.log` means the file will be named `app.log`

📝 Example:

- If your project is at:
- `C:\Users\Vijay\Projects\MySpringApp`
-
- Then the log file will be at:
- `C:\Users\Vijay\Projects\MySpringApp\logs\app.log`

❗ Important Notes:

- No need to create the `logs` folder manually — Spring Boot will auto-create it.
- If the app doesn't have write permission, the file won't be created.

🔧 Logging Hierarchy (from highest to lowest verbosity)

Level	Description	Use Case
OFF	Disables all logging.	Used to completely turn off logging.
ERROR	Logs only serious errors that may cause the application to fail.	Application crashes, connection failures.
WARN	Logs potential issues or warnings, but app still runs.	Deprecated API usage, configuration issues.
INFO	Logs general application flow and important runtime events.	App startup, shutdown, REST endpoint access.
DEBUG	Logs detailed debug info for developers.	Variable values, method entries/exits.
TRACE	Logs very fine-grained and low-level info.	Deep internal flow tracing.
ALL	Enables logging at all levels (TRACE and above).	Rarely used; mostly for enabling everything.

🎯 Practical Example:

```
logger.error("Something failed");
logger.warn("This might be an issue");
logger.info("Service started");
logger.debug("Value of X = " + x);
logger.trace("Entering method foo()");
```

- If level is set to `INFO`, only `INFO`, `WARN`, and `ERROR` logs will appear.
- If the level is set to `DEBUG`, you will see `DEBUG`, `INFO`, `WARN`, and `ERROR`

✅ Recommendation:

Environment	Recommended Log Level
Development	`DEBUG` or `TRACE`
Testing	`INFO` or `DEBUG`
Production	`INFO` or `WARN`

✅ Advantages:

- Easy configuration.
- Built-in support with profiles.
- File and console logging supported.
- Can change log level at runtime with actuator.

❌ Disadvantages:

- Verbose logs unless filtered.
- No log rotation by default (need config).
- No GUI for viewing logs.

💡 Log Monitoring Tools:

- **Log4j2**: Better performance.
- **ELK Stack** (ElasticSearch + Logstash + Kibana): For centralized log analysis.
- **Sentry**, **Splunk**, **Datadog**: For monitoring/log aggregation.

Importance interview question:

Q1: Where do we configure log appenders in Spring Boot (real-time projects)?

Answer:
In real-time Spring Boot projects, we configure **log appenders** inside the
➡ `logback-spring.xml` file (placed in `src/main/resources` folder).

This file allows us to define how and where logs are stored — like **console, file, or rolling files**.

Q2: What is an Appender in logging?

Answer:
An **Appender** decides **where the log messages will go**, such as:

- Console (using `ConsoleAppender`)
- File (using `FileAppender`)
- Rolling files (using `RollingFileAppender`)

Q3: How do we define log format in real-time projects?

Answer:
We define the **log message format** inside the `<encoder>` tag in `logback-spring.xml`.

```
<encoder>
   <pattern>%d{yyyy-MM-dd HH:mm:ss} %-5level [%thread] %logger{36} -
%msg%n</pattern>
</encoder>
```

This pattern controls how each log line looks (time, level, thread, class, message).

Q4: How do we implement log rolling (log rotation)?

Answer:
Log rolling (or rotation) is configured using **`RollingFileAppender`** with a **`TimeBasedRollingPolicy`**.
It automatically creates new log files (daily or by size) and deletes old ones.

Example:

```
<rollingPolicy
class="ch.qos.logback.core.rolling.TimeBasedRollingPolicy">
   <fileNamePattern>logs/app-%d{yyyy-MM-dd}.log</fileNamePattern>
   <maxHistory>7</maxHistory>
</rollingPolicy>
```

- Creates a new log file every day
- Keeps only the last **7 days** of logs

Q5: Why do we use RollingFileAppender in real-time projects?

Answer:
Because it:

- Prevents single log files from getting too large 🧹
- Automatically rotates logs daily or by file size 📅
- Helps in log management and troubleshooting 🧠

Q6: How do we configure a separate error log file (error.log) in real-time Spring Boot projects?

Answer:
In real-time projects, we often store **error logs separately** for easier debugging and production monitoring.

This is done using a **dedicated `RollingFileAppender`** for errors in the `logback-spring.xml` file.

🗒️ Example Configuration

```
<appender name="ERROR_FILE"
class="ch.qos.logback.core.rolling.RollingFileAppender">
    <file>logs/error.log</file>

    <rollingPolicy
class="ch.qos.logback.core.rolling.TimeBasedRollingPolicy">

<fileNamePattern>logs/error-%d{yyyy-MM-dd}.log</fileNamePattern>
        <maxHistory>10</maxHistory>
    </rollingPolicy>

    <encoder>
        <pattern>%d{yyyy-MM-dd HH:mm:ss} %-5level [%thread]
%logger{36} - %msg%n</pattern>
    </encoder>

    <!-- ✅ Only capture ERROR level logs -->
    <filter class="ch.qos.logback.classic.filter.LevelFilter">
        <level>ERROR</level>
        <onMatch>ACCEPT</onMatch>
        <onMismatch>DENY</onMismatch>
    </filter>
</appender>

<!-- Attach this appender to the root logger -->
<root level="INFO">
    <appender-ref ref="CONSOLE" />
    <appender-ref ref="FILE" />
    <appender-ref ref="ERROR_FILE" />
</root>
```

💡 How it Works

- All logs (INFO, WARN, ERROR) go to:
    - Console
    - Main `app.log` file
- Only **ERROR level logs** are written to:
    - `error.log` file

So you'll have two files in `/logs`:

- `app-2025-11-09.log` → all logs
- `error-2025-11-09.log` → only errors

THANK YOU AND FOLLOW ME FOR MORE