AXON ACTIVE
Focusing on your agility

Axon Active Vietnam Co., Ltd.
Hai Au Building, 39B Truong Son St.
Ward 4, Tan Binh District
Ho Chi Minh City
Vietnam
www.axonactive.com
info@axonactive.com

Axon Active Schweiz AG
Schlössli Schönegg
Wilhelmshöhe
6003 Luzern
Switzerland
www.axonactive.ch
info@axonactive.ch

Other Branches
Da Nang
214 30/4 Street, Hai Chau District, Da Nang, Vietnam
Can Tho
57-59A CMT8 Street, Ninh Kieu District, Can Tho, Vietnam
San Francisco
281 Ellis Street, San Francisco, California 94102, USA

Guide to Log4J2 - Logging Rules for Agile Course

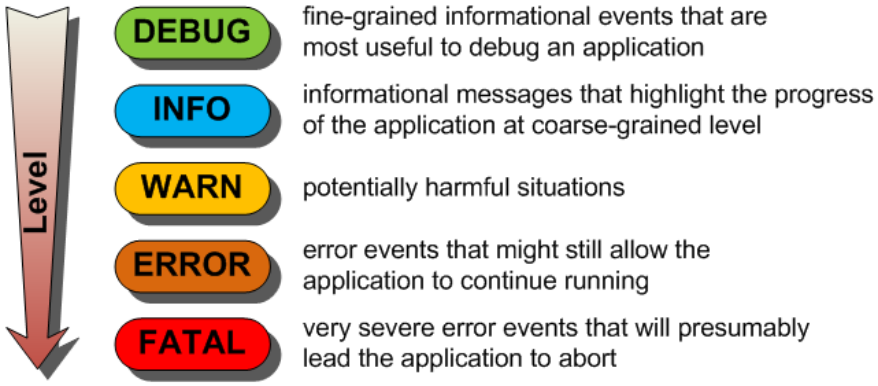
Author: *VAMOS Team*
Date: January 19, 2024

Contents

I. Introduction

Understand Logging

- Logging is the process of recording information, events, or messages generated during the execution of a program.
- Log Levels represent the severity or importance of a log message. Common levels include:



What is Log4J2?

- Log4j 2 is a Java-based logging utility. It provides a flexible and efficient logging framework for Java applications.

II. Local Installation

1/ Add LOG4J-core and LOG4J-api libraries into the POM.xml file.

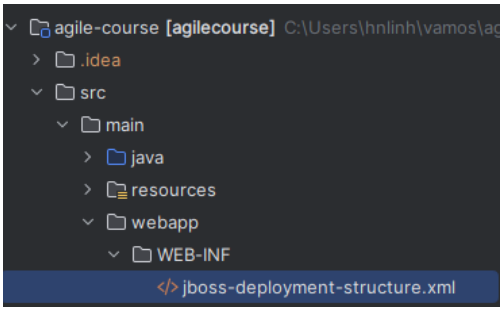
Please note that you should use Log4J versions 2.x.x, since the 1.x.x versions do not support Java8 and have low performance.

```
<!-- https://mvnrepository.com/artifact/org.apache.logging.log4j/log4j-core -->
<dependency>
    <groupId>org.apache.logging.log4j</groupId>
    <artifactId>log4j-core</artifactId>
    <version>2.19.0</version>
</dependency>

<!-- https://mvnrepository.com/artifact/org.apache.logging.log4j/log4j-api -->
<dependency>
    <groupId>org.apache.logging.log4j</groupId>
    <artifactId>log4j-api</artifactId>
    <version>2.19.0</version>
</dependency>
```

- **LOG4J-CORE:**
- The `log4j-core` library contains the actual implementation of the Log4j 2 logging framework.
- It includes the runtime components responsible for processing and handling log events.
- **LOG4J-API:**
- The `log4j-api` (Logging API) provides the interface or API through which your application interacts with the logging framework.
- It defines the logging methods and structures, such as Logger, Appender, Layout, and other core logging concepts.

2/ Inside the Main folder, add webapp/WEB-INF/jboss-deployment-structure.xml



WEB-INF folder serves as a deployment descriptor location and a secure area for sensitive configuration files, resources, and classes.

Add the code below inside the file:

```
<?xml version="1.0" encoding="utf-8"?>
<jboss-deployment-structure>
    <deployment>
        <!-- Exclusions allow you to prevent the server from automatically adding some dependencies -->
        <exclude-subsystems>
            <subsystem name="logging" />
        </exclude-subsystems>
    </deployment>
</jboss-deployment-structure>
```

The code below helps to exclude the logging subsystem. We need this to avoid the default logging setup of JBoss/WildFly.

3/ Inside Resource folder, set up Log4j2.xml:

This file controls how logging is configured and where log messages will be written.

```
<?xml version="1.0" encoding="UTF-8"?>
<Configuration status="WARN">
   <Properties>
       <!--Logging Properties-->
       <Property name="LOG_PATTERN">%-5p | %d{yyyy-MM-dd HH:mm:ss} |
  %m%n</Property>
       <Property name="LOG_ROOT">${env:JBOSS_HOME}/customization/log</Property>
   </Properties>

   <Appenders>
       <!-- Console Appender -->
       <Console name="console" target="SYSTEM_OUT">
           <PatternLayout pattern="${LOG_PATTERN}" />
       </Console>

       <!-- Rolling File Appender -->
       <RollingFile name="agilecourse" fileName="${LOG_ROOT}/agilecourse.log"

  filePattern="${LOG_ROOT}/agilecourse-%d{yyyy-MM-dd}-%i.log">
           <LevelRangeFilter minLevel="FATAL" maxLevel="INFO" onMatch="ACCEPT"
  onMismatch="DENY"/>
           <PatternLayout pattern="${LOG_PATTERN}" />
           <Policies>
               <TimeBasedTriggeringPolicy interval="1" modulate="true" />
               <SizeBasedTriggeringPolicy size="10MB"/>
           </Policies>
           <DefaultRolloverStrategy>
               <Delete basePath="${LOG_ROOT}">
```

```
                <IfLastModified age="14d" />
            </Delete>
        </DefaultRolloverStrategy>
    </RollingFile>
  </Appenders>

  <Loggers>
    <Logger name="com.axonactive.agilecourse" level="info"
 additivity="false">
        <AppenderRef ref="agilecourse" />
        <AppenderRef ref="console" />
    </Logger>

    <!-- Root Logger Configuration -->
    <Root level="all" additivity="false">
        <AppenderRef ref="agilecourse" />
        <AppenderRef ref="console" />
    </Root>
  </Loggers>
</Configuration>
```

Let's take a look into the details of this file.

<Properties> section:

- `LOG_PATTERN`: Specifies the pattern for log messages. It includes placeholders for log level (`%-5p`), timestamp (`%d{yyyy-MM-dd HH:mm:ss}`), and a custom property named `mail` (`%X{mail}`), followed by the log message itself (`%m%n`).
- `LOG_ROOT`: Specifies the root directory for log files. It uses an environment variable `JBOSS_HOME` to dynamically set the log directory within the JBoss/WildFly server's home directory.

```
<Properties>
    <!--Logging Properties-->
    <Property name="LOG_PATTERN">%-5p | %d{yyyy-MM-dd HH:mm:ss} | %X{mail} | %m%n</Property>
    <Property name="LOG_ROOT">${env:JBOSS_HOME}/customization/log</Property>
</Properties>
```

<Appenders> section:

- The appenders is the layout you want the log messages to be printed out. For example, you can write them in Console, various types of Files, and even in Database Scripts.

- **Console Appender (`console`):**
- Outputs log messages to the console (standard output).
- Uses the specified `LOG_PATTERN` for formatting.

```
<Appenders>
    <!-- Console Appender -->
    <Console name="console" target="SYSTEM_OUT">
        <PatternLayout pattern="${LOG_PATTERN}" />
    </Console>
```

- **Rolling File Appender (`agilecourse`):**
- Outputs log messages to a rolling log file (`agilecourse.log`).
- Uses a specified file pattern for rolling log files based on date and index.
- Applies a `LevelRangeFilter` to filter log messages between FATAL and INFO levels.
- Specifies a pattern for log messages using the `LOG_PATTERN`.
- Implements log file rotation policies:
- `TimeBasedTriggeringPolicy`: Rolls over log files daily and modulates filenames.
- `SizeBasedTriggeringPolicy`: Rolls over log files when they reach 10MB in size.
- Uses `DefaultRolloverStrategy` to delete log files older than 14 days.

```
<!-- Rolling File Appender -->
<RollingFile name="agilecourse" fileName="${LOG_ROOT}/agilecourse.log"
             filePattern="${LOG_ROOT}/agilecourse-%d{yyyy-MM-dd}-%i.log">
    <LevelRangeFilter minLevel="FATAL" maxLevel="INFO" onMatch="ACCEPT" onMismatch="DENY"/>
    <PatternLayout pattern="${LOG_PATTERN}" />
    <Policies>
        <TimeBasedTriggeringPolicy interval="1" modulate="true" />
        <SizeBasedTriggeringPolicy size="10MB"/>
    </Policies>
    <DefaultRolloverStrategy>
        <Delete basePath="${LOG_ROOT}">
            <IfLastModified age="14d" />
        </Delete>
    </DefaultRolloverStrategy>
</RollingFile>
```

<Loggers> section:

- This section defines how logs are written and handled based on each package of the project.
- The RootLogger serves as the root of the logger hierarchy.It acts as a fallback for log messages that don't have a more specific logger to handle them.
- In package ***com.axonactive.agilecourse***, we configure a logger with these properties:
- Set the logging level to `info`.
- `additivity="false"` ensures that log events are not propagated to the root logger.

```
<Loggers>
    <Logger name="com.axonactive.agilecourse" level="info" additivity="false">
        <AppenderRef ref="agilecourse" />
        <AppenderRef ref="console" />
    </Logger>

    <!-- Root Logger Configuration -->
    <Root level="all" additivity="false">
        <AppenderRef ref="agilecourse" />
        <AppenderRef ref="console" />
    </Root>
</Loggers>
```

Note: By default log4j2 logging is additive. It means that all the parent loggers will also be used when a specific logger is used.

In this project, we set it to false.

III. Implement logging into the project

Add logger into the class you want to write log inside:

```
import org.apache.logging.log4j.LogManager;
import org.apache.logging.log4j.Logger;
```

```
public class UniversityResource {
    3 usages
    private static final Logger logger = LogManager.getLogger(UniversityResource.class);
```

Write log using loggers:

```
public Response getAllUniversities() {
    logger.info("Attempting to get Universities List...");
    List<University> universityList = new ArrayList<>();
    try {
        universityList = universityService.getAll();
        logger.info("Get University List successfully.");
    } catch (Exception e) {
        logger.error( message: "Error while retrieving Universities List", e);
        e.printStackTrace();
    }
    return Response.ok().entity(universityList).build();
}
```

When we call the APIs, we will see something like this:

1/ Console log

```
10:10:46,711 INFO  [stdout] (default task-1) INFO  2024-02-15 10:10:46 UniversityResource - Attempting to get Universities List...
10:10:46,712 INFO  [stdout] (default task-1) Hibernate: select university0_.id as id1_3_, university0_.created_at as created_2_3_, univer
10:10:46,715 INFO  [stdout] (default task-1) INFO  2024-02-15 10:10:46 UniversityResource - Get University List successfully.
```

2/ File log

```
INFO  2024-02-15 10:43:33 UniversityResource - Attempting to get Universities List
INFO  2024-02-15 10:43:33 UniversityResource - Get University List successfully.
```

IV. Logging Rules For Agile Course Project

1/ Avoid Excessive Logging

- Logging too much information can cause:
  - Too much space required to store all the log files;
  - You may end up with too much information without being able to find out the root solutions.
- In order to prevent the mentioned problems:
  - **Log at appropriate level:** Common levels include: DEBUG, INFO, WARN, and ERROR;
  - **Log only essential information** that is required for debugging or monitoring;
  - **Manage log file sizes**, and remove the files periodically, so they won't take too much disk space *(This approach was already implemented in the AgileCourse project)*.

2/ Logging Pattern *(The pattern was formatted in log4j2.xml file of AgileCourse project)*

%-5p %d{yyyy-MM-dd HH:mm:ss} %c{1} - %m%n

3/ Logging Message Convention

- Write meaningful messages;
- Use parameterized messages (such as "{}"):

```
logger.info( message: "Attempting login for user '{}'", getMaskedEmail(loginDTO.getEmail()));
```

4/ Be careful with sensitive information

- Avoid logging sensitive information. You can either not display it or mask them like this:

```
INFO  2024-02-15 10:22:03 AuthResource - Attempting login for user 'huy.n*********'
Hibernate: select user0_.id as id1_4_, user0_.created_at as created_2_4_, user0_.updated_at as updated_3_
INFO  2024-02-15 10:22:03 AuthResource - Validate login credential successfully.
INFO  2024-02-15 10:22:03 AuthResource - Generating token...
INFO  2024-02-15 10:22:03 AuthResource - Generate token successfully.
```

V. Logging usecases:

1. **Error and Exception Tracking:** Trace back logs pinpoint the specific line of code and surrounding context where an error occurred, aiding in quick resolution.

2. **Security Auditing:** By logging security-related events such as authentication attempts can track user activities and detect potential security breaches.

3. **Troubleshooting Production Issues:** Logs from deployed systems provide critical information for diagnosing and fixing errors impacting users.

VI. Best practices for logging:

- Declare the logger to be both static and final to ensure that every instance of a class shares the common logger object.

- Avoid logging at 'every' place where a custom exception is thrown. Log in the ExceptionMapper Classes

```java
@Provider
public class IOExceptionMapper implements ExceptionMapper<IOException> {
    1 usage
    private static final Logger logger = LogManager.getLogger(IOException.class);

    no usages
    @Override
    public Response toResponse(IOException e) {
        ResponseBody responseBody = new ResponseBody(Response.Status.BAD_REQUEST, KEY_FILE_NOT_FOUND, FILE_NOT_FOUND);

        StackTraceElement[] stackTraceArray = e.getStackTrace();
        String logMessage = String.format("%s:%d - %s",
                stackTraceArray[0].getClassName(),
                stackTraceArray[0].getLineNumber(),
                responseBody.getErrorMessage());

        logger.error(logMessage);

        return Response.status(Response.Status.BAD_REQUEST)
                .entity(responseBody)
                .build();
    }
}
```

- Should log the INPUT for Validation. Because we need to know what people input that causes errors. (**DO NOT** log sensitive information: e.g: password, email should be masked)

- Depending on the type of exception, we log the stack trace or not. For instance, if the exception is too general, we should trackstace to investigate the issue. The log stack trace only make the file log size bigger.

- Any other intermediate redundant logging statements, which are used just for the purpose of debugging can still be avoided.

Example

```
try {
   LOGGER.debug("About to enter getItemDescription method");
   // The above logging statement is not required,
   // if getItemDescription() method logs its method entry

   String itemDesc = getItemDescription(itemNumber);

   LOGGER.debug("Exited getItemDescription method");
   // The above logging statement is not required,
   // if getItemDescription() method logs its method exit

} catch (ApplicationCustomException ace) {
   LOGGER.error(ace.getErrorMessage());
   throw se;
}
```

References:

https://stackoverflow.com/questions/7839565/logging-levels-logback-rule-of-thumb-to-assign-log-levels

https://romanglushach.medium.com/java-rest-api-logging-best-practices-and-guidelines-bf5982ee4180

https://dev.to/gauthierplm/how-to-output-log4j2-logs-as-json-5an3