Adding Logs to Test Cases (Log4j2)

- Understanding Logs:
  Log is a record of events or actions performed during the execution of test cases, stored in the form of text — this process is called **logging**.
- Example Scenario:
  Suppose we are executing our test cases that perform several actions such as **launching the application**, **logging in**, and **user registration**.
  → By recording these events in the form of text messages, we create a **log file**

Advantages of Generating Log Files

- There are two kinds of logs: **application logs** and **automation logs**.
  Any action or event performed by a user or customer on the application is recorded in a log file at the backend.
- In real-time (production) applications, if a user performs any **unauthorized activity**, the system can easily track it using the log file. This helps the team take the necessary actions such as **blocking or holding the account**.
- Logs act as a **security mechanism** and provide visibility into system and user activities.
- **Automation logs** record all events that occur during the execution of automated test cases.
  Sometimes, a defect may be reproducible in the QA environment but not in the development environment. In such cases, developers can review the **log file** to analyze the issue and identify the root cause.

Types of Logs

Different log levels (from lowest to highest severity): **TRACE** < **DEBUG** < **INFO** < **WARN** < **ERROR** < **FATAL**

- **TRACE** – The most detailed logging level. Captures every small step in the program's execution (*e.g.* method entry/exit, variable values).
- **DEBUG** – Detailed technical information for debugging (*e.g.* API calls, data processing, configuration loading).
- **INFO** – General information about the execution flow (*e.g.* "Test started", "User logged in", "Registration successful").
- **WARN** – Highlights potential issues that don't stop the execution but might cause problems later (*e.g.* Low disk space, Deprecated API used).
- **ERROR** – Captures error messages when something fails during execution (*e.g.* Unable to connect to database, Element not found)
  Example: "Unable to connect to database" or "Element not found."
- **FATAL** – Indicates severe issues that cause the unexpected termination of the application or test (*e.g.* System crash, Critical configuration missing).

Note: The **DEBUG** log level is mostly used and understood by developers, as it captures detailed information about all transactions between the **application (client)** and the **backend system (server)**.

The **INFO** log level is commonly used and understood by testers, as it records general information and displays all the messages written in the test cases within the log file.

Log4j2 Terminologies and Setup

- There are two key terminologies in Log4j2:
  1. **Appenders** – Define **where** to generate logs (*e.g.* Console, File).
     Example: Log messages can be written to a file or displayed in the console window.
  2. **Loggers** – Define **what type of logs** to generate (*e.g.* TRACE < DEBUG < INFO < WARN < ERROR < FATAL).
- To generate logs for automation tests using Log4j2:
  1. Add the required **Log4j2 dependencies** in the `pom.xml` file.
  2. Add the `log4j2.xml` **configuration file** under `src/test/resources`.
     This XML file defines the loggers, appenders, and logging levels. A template configuration file is provided by Apache Log4j2.

Logger Configuration

In the **Loggers** section of the `log4j2.xml` file:

- Specify the **log level** (TRACE, DEBUG, INFO, WARN, ERROR, FATAL) to determine which types of log messages should be generated.
- Associate the logger with an **Appender** (*e.g.* File or Console) using the `AppenderRef` tag, which decides **where** the log messages will be recorded.

```
<Loggers>
    <Root level="INFO">
        <!--<AppenderRef ref="Console"/>-->
        <AppenderRef ref="File"/>
    </Root>
</Loggers>
```

Specifying the Log File Path

In the **Properties** section of `log4j2.xml`:

- generate log files inside a `logs` folder in our project and assign the folder path to the property `basePath`.

```
<Properties>
    <Property name="log-path">./logs</Property>
    <Property name="log-pattern">%d{yyyy-MM-dd HH:mm:ss} %-5p %c{1}:%L - %m%n</Property>
</Properties>
```

Appenders Configuration

In the **Appenders** section of `log4j2.xml`:

- Assign the same value of the `name` attribute as a variable to the `fileName` attribute in the `RollingFile` appender, followed by the name of the log file to be created "`automation.log`".

➢ Add a **timestamp** to the value of the `filePattern` attribute in the `RollingFile` appender to:
  - Differentiate between multiple generated log files in the `logs` folder.
  - Automatically archive and maintain old log files when the log file size exceeds the defined limit.

```
<Appenders>
    <Console name="Console" target="SYSTEM_OUT">
        <PatternLayout pattern="${log-pattern}"/>
    </Console>

    <RollingFile name="File" fileName="${log-path}/automation.log"
                 filePattern="${log-path}/automation-%d{MM-dd-yyyy}-%i.log.gz">
        <PatternLayout pattern="${log-pattern}"/>
        <Policies>
            <TimeBasedTriggeringPolicy/>
            <SizeBasedTriggeringPolicy size="1MB"/>
        </Policies>
        <DefaultRolloverStrategy max="10"/>
    </RollingFile>
</Appenders>
```

Integrating Logging into the BaseClass

– Update the `setup()` method in the reusable `BaseClass` to include the logging configuration steps, as the generation of log messages is part of the common setup activities.
– Load the `log4j2.xml` configuration using the `getLogger()` method provided by the `LogManager` class. The file will be automatically fetched from the `resources` folder based on the default classpath configuration.
– Use the `getLogger()` method from the predefined `LogManager` class and specify the class name from which you want to generate logs: The expression `this.getClass()` ensures that, at runtime, the logger dynamically takes the name of the test class being executed.
– The resulting `logger` object (variable) will be used during test execution to create and record log messages.

```
public class BaseClass {

    public WebDriver driver;
    public Logger logger;

    @BeforeClass
    public void setup() {
        logger = LogManager.getLogger(this.getClass());
```