Spring Boot

Workshop N°2

Objectives

- Understand the importance of **logging** in a Spring Boot application.
- Learn how to configure and use logging in Spring Boot.
- Implement custom logic using the **CommandLineRunner** interface.

Task 1:

1. Create a new Spring Boot project named **"LoggingProject"** using STS without starters.
2. Add the following metadata to your project's build configuration:
   - Project Type: Maven
   - Group Id: com.example
   - Artifact Id: logging
3. Create a new class named **"MyApplicationRunner"** in the appropriate package and implement the **CommandLineRunner** interface.
4. In the **run** method of the MyApplicationRunner class, log the messages "**Application started**" and "**Application finished**" using the **LOGGER** object with the **log level info**.
5. Run the Spring Boot application and observe the console output. You should see the log messages printed when the application starts and finishes.

```
2023-05-23T23:57:44.096+01:00  INFO 9628 --- [  restartedMain] c.example.logging.MyApplicationRunner    : Application started
2023-05-23T23:57:44.096+01:00  INFO 9628 --- [  restartedMain] c.example.logging.MyApplicationRunner    : Application finished
```

6. Add a simple calculation that divides a number by zero inside the run method. Observe how the application handles the division by zero error and logs an appropriate **ERROR** message.
7. Re-Run the Spring Boot application and observe new the log messages in the console.

```
2023-05-23T23:55:08.411+01:00  INFO 3632 --- [  restartedMain] c.example.logging.MyApplicationRunner    : Application started
2023-05-23T23:55:08.419+01:00 ERROR 3632 --- [  restartedMain] c.example.logging.MyApplicationRunner    : Error occurred: Division by zero
2023-05-23T23:55:08.469+01:00  INFO 3632 --- [  restartedMain] c.example.logging.MyApplicationRunner    : Application finished
```

8. Modify the **logging.level** property in the **application.properties** file to set the logging level for all dependencies to **WARN**. Then, re-run your application and observe the changes.
9. Configure your application to write the log messages to a file named "**mylog.log**" inside the "**logs**" directory. Re-Run your application and check if the log messages are being properly written to the file.
10. The default log pattern in Spring Boot is as follows:

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

    This pattern consists of the following placeholders:
    - %d{yyyy-MM-dd HH:mm:ss}: Represents the date and time of the log message in the format "yyyy-MM-dd HH:mm:ss".
    - [%thread]: Represents the name of the thread that generated the log message.
    - %-5level: Represents the log level with a width of 5 characters, left-aligned.
    - %logger{36}: Represents the name of the logger generating the log message, truncated to a maximum length of 36 characters.
    - %msg: Represents the log message itself.
    - %n: Represents a newline character, indicating the end of the log message.

You can customize the "**Logging pattern**" by modifying the following properties:

- logging.pattern.console: Change the logging pattern for console output.
- logging.pattern.file: Change the logging pattern for file output.

a. Change the logging pattern for the console by reducing the number of characters for the logger to 20 and removing the PID from the display

b. Change the logging pattern for the log file to display the date as day/month/year and the logger with a width of 100 characters.

Task 2: [Optional]

In many systems and applications, it is common to generate log files that record important activities and events. These log files are crucial for debugging, monitoring, and performance analysis of the system. A real use case of logs is copying log files at the end of each week.

Write a method named **copyLogFile** in the MyApplicationRunner class. This method should be responsible for copying the log file from the "logs/mylog.log" location to the "logs/old/mylog.log" location.

The method should perform the following steps:

1. Create a **File** object to represent the source log file "logs/mylog.log".
2. Create a **File** object to represent the destination log file "logs/old/mylog.log".
3. Check if the source log file exists. If it doesn't, create the necessary parent directories and the log file itself.
4. Check if the destination log file exists. If it doesn't, create the necessary parent directories and the log file itself.
5. Use the **Files.copy()** method to copy the source file to the destination file, replacing any existing file if necessary.
6. Log an info message indicating that the log file has been successfully copied, along with the destination file path.
7. If any exception occurs during the file copying process, log an error message with the specific exception message.

Additionally, make sure to call the **copyLogFile** method within the **run** method of the **MyApplicationRunner** class to execute it when the application starts.

Here are the links to the documentation that can help you understand the code:

1. **File class documentation**: Provides information about the File class, which represents a file or directory path in the file system.
2. **Files class documentation**: Provides information about the Files class, which contains various utility methods for working with files, including file copying.
3. **StandardCopyOption class documentation**: Provides information about the StandardCopyOption class, which defines the options for file copy operations