Arthrex – Coding Assignment

Prabhjot Singh

+91 8607682303

Prabhjotchugh0805@gmail.com

Documentation

The Logger module is designed to collect and manage logs in a multithreaded environment. It meets the following requirements:

1. Inserting Log Messages:

Log messages are inserted into a file at a predefined location using a RotatingFileHandler from the logging.handlers module.

2. Log Format:

Each log message is saved in the log file in the following format:

- Timestamp: The time at which the log message is generated.
- Log Level: Indicates the severity level of the log message (DEBUG, INFO, ERROR).
- File Name: The source code file from which the log message was created.
- Function Name: The name of the function from which the log message was invoked.
- Thread ID: The ID of the thread that generated the log message.
- Actual Log Message: The content of the log message.

3. Log File Rotation:

- The log file is rotated after every 5 MB to prevent it from growing too large.
- A limit of 10 rotations is imposed, after which the old logs are automatically wrapped around.

Explanation of Code Components:

The **Logger class** is responsible for initializing and managing the logging functionality.

- The __init__() method initializes the logger with a RotatingFileHandler, specifying the log file name, maximum file size, and backup count.
- The log() method logs messages at different levels (DEBUG, INFO, ERROR) and formats them according to the specified format.

The **test_logging()** function is used to simulate multithreaded logging for testing purposes.

- It creates multiple threads, each of which generates log messages at different levels to test the logger's functionality.

Example Usage:

```
# Initialize the logger

logger = Logger("app.log", max_bytes=5*1024*1024, backup_count=10)


# Simulate multithreaded logging for testing

test_logging(logger)
```

Testing

Test cases to automate the testing of the logger module.

1. Single Threaded Logging:

- Create a Logger instance.
- Log messages at different levels (DEBUG, INFO, ERROR) using the logger instance.
- Read the log file and verify that each logged message appears in the correct format and contains the expected content.

2. Multithreaded Logging:

- Create a Logger instance.
- Define a worker function that logs messages at different levels (DEBUG, INFO, ERROR).
- Create multiple threads, each executing the worker function.

- Wait for all threads to complete.
- Read the log file and verify that all logged messages from different threads appear in the correct format and contain the expected content.

3. Log Rotation:

- Create a Logger instance with a small maximum file size and backup count.
- Log messages continuously until the log file exceeds the maximum size.
- Verify that the log file is rotated and the old log file is renamed correctly.
- Repeat logging messages until the maximum number of rotations is reached.
- Verify that the old log files are wrapped around and new log messages are correctly logged.

4. Log Format:

- Create a Logger instance.
- Log messages at different levels (DEBUG, INFO, ERROR) using the logger instance.
- Read the log file and verify that each logged message follows the specified format, including the timestamp, log level, file name, function name, thread ID, and actual log message.

5. Error Handling:

- Attempt to log messages with an invalid log level (e.g., "INVALID_LEVEL").
- Verify that a ValueError is raised when attempting to log messages with an invalid log level.

6. Integration Testing:

- Integrate the Logger module into a sample application or codebase.
- Log various messages at different levels during the execution of the application.
- Inspect the log file generated by the Logger module and verify that all logged messages are present and correctly formatted.

By implementing and automating these test cases, the Logger module functions correctly and reliably in various scenarios, including single-threaded and multithreaded environments, log rotation, error handling, and integration with other codebases.