🧾 Flask Logging Assignment Documentation

📘 Title:

Implementation of Logging in Flask Using Different Severity Levels

---

🎯 Objective

The purpose of this assignment is to demonstrate the use of Flask's built-in logging system by creating multiple API endpoints, each representing a specific logging severity level.
The goal is to understand how to generate and manage application logs with appropriate severity and to simulate real-world scenarios where each log type would apply.

🏗️ Project Overview

Students are required to create a Flask web application that uses the built-in Python logging module. The application must define several endpoints (`/debug`, `/info`, `/warning`, `/error`, and `/critical`) — each designed to trigger and log messages corresponding to a specific severity level.

Each endpoint should simulate a realistic scenario matching its severity type. The logs should appear in both the **console** and a **log file** (e.g., `app.log`).

⚙️ Expected Application Behavior

1. **Logging Configuration**
    - The application must include a proper logging configuration that defines log format, handlers, and log levels.

- Logs should be printed to both:
    - The **console** (for real-time viewing)
    - A **log file** (for record-keeping)
- Each log entry must include a timestamp, log level, and message.

2. **Endpoints and Severity Simulation**
    - Each endpoint corresponds to one of the five log severity levels.
    - Each should simulate a scenario and return a clear response message related to the log context.

🔸 Endpoint Descriptions and Scenarios

1. `/debug`

- **Purpose:** Used to simulate detailed diagnostic information.
- **Scenario:** Represents a situation where a developer is tracing the flow of execution or debugging issues during development.
- **Behavior:** The endpoint should log a message indicating that it is used for debugging purposes and return a response confirming diagnostic activity.
- **Severity Level:** `DEBUG`
- **When to Use:**
  Use when troubleshooting, testing logic flow, or inspecting variables — primarily during the development phase.

---

2. `/info`

- **Purpose:** Used to indicate normal operation messages and system status.
- **Scenario:** Represents a normal event such as the application running smoothly or a successful API request.
- **Behavior:** The endpoint should log an informational message that the system is operating normally and return a confirmation response.
- **Severity Level:** `INFO`
- **When to Use:**
  Use to record standard operational messages, such as application start-up, user access, or routine operations.

---

3. `/warning`

- **Purpose:** Indicates potential issues or unusual circumstances that may not yet cause errors.
- **Scenario:** Simulates a situation where a configuration issue, performance concern, or deprecated feature might cause future problems.
- **Behavior:** The endpoint should log a warning message describing a potential issue and return a response advising caution.
- **Severity Level:** `WARNING`
- **When to Use:**
  Use when an action could lead to a problem but doesn't stop the application (e.g., low memory, nearing a limit, or deprecated usage).

---

4. `/error`

- **Purpose:** Represents errors that prevent part of the application from functioning properly.
- **Scenario:** Simulates an exception, such as a division by zero or a failed database connection.
- **Behavior:** The endpoint should handle the simulated error gracefully, log the error message, and return a response stating that an error occurred.
- **Severity Level:** `ERROR`
- **When to Use:**
  Use when an operation fails due to a runtime exception, missing resource, or failed process, but the overall system can still continue running.

---

5. `/critical`

- **Purpose:** Used for serious system failures or events requiring immediate attention.
- **Scenario:** Simulates a major system failure — for example, a fatal dependency issue or data corruption.
- **Behavior:** The endpoint should log a critical-level message indicating a severe failure and return a response stating that a critical issue occurred in the application.
- **Severity Level:** `CRITICAL`
- **When to Use:**
  Use for catastrophic issues that may cause the entire system or a key component to stop functioning (e.g., security breach, critical data loss).

🧠 Summary of Severity Levels

Severity Level	Description	Typical Usage
DEBUG	Provides detailed diagnostic information.	For developers to trace internal flow or identify bugs.
INFO	Confirms that things are working as expected.	For standard operational updates.
WARNING	Indicates potential issues that might need attention.	For recoverable issues or configuration risks.
ERROR	Application error that interrupts functionality.	For caught exceptions or failed operations.
CRITICAL	Indicates severe, system-wide failure.	For fatal events that require immediate investigation.

🧾 Expected Output

Each endpoint should return a JSON response describing the situation, while the log file should record a timestamped entry with the corresponding log level and message.

Example Response and Log Summary:

Endpoint	Example Response	Example Log Output
`/debug`	“Debugging mode active. Used for diagnostic purposes.”	`[2025-10-06 10:00:00] DEBUG in app: Debugging endpoint hit - diagnostic log.`
`/info`	“App is running smoothly.”	`[2025-10-06 10:01:00] INFO in app: Information endpoint accessed successfully.`
`/warning`	“Warning: Possible issue detected.”	`[2025-10-06 10:02:00] WARNING in app: Configuration issue detected.`
`/error`	“An error occurred during processing.”	`[2025-10-06 10:03:00] ERROR in app: Division by zero exception occurred.`

`/critical`	“An error occurred in the application.”	`[2025-10-06 10:04:00] CRITICAL in app: Critical failure encountered - immediate attention required.`