**Django Logging: StepbyStep Guide with Implementation**

*Ref:* [*https://docs.djangoproject.com/en/5.1/topics/logging/*](https://docs.djangoproject.com/en/5.1/topics/logging/)

Logging is a crucial aspect of any application, especially for debugging and error tracking. Django provides a flexible and powerful logging framework based on Python's builtin `logging` module.

**Step 1: Understand Django's Logging Architecture**

**Loggers:** Define the source of log messages. Each logger is associated with a specific part of your application, like `django`, `django.request`, or custom apps like `student`.

**Handlers:** Specify where the log messages are sent. Handlers can send logs to the console, a file, email, or even external services.

**Formatters:** Define the structure of the log messages (e.g., timestamp, log level, message content).

**Filters:** Provide finegrained control over which log messages are processed by the logger.

**Step 2: Basic Logging Setup in Django**

1. Open the **`settings.py`** file: The logging configuration is usually done in this file.

2. Add or update the **`LOGGING`** dictionary: This is where you configure all your loggers, handlers, and formatters. Refer to setting.py

**Step 3: Explanation of the Configuration**

**1. Version:**

The `version: 1` tells Django the version of the logging configuration schema.

**2. Disable Existing Loggers:**

`disable\_existing\_loggers: False` ensures that Django’s builtin loggers (such as for the `django` and `django.request` components) are not disabled.

**3. Formatters:**

`verbose`: Logs detailed information like log level, timestamp, module, and message.

`simple`: A simpler format, showing only the log level and message.

**4. Handlers:**

`file`: Logs messages to a file (`django\_app.log`). It uses the `verbose` formatter, so log messages will include detailed information.

`console`: Sends log messages to the terminal using the `simple` formatter.

**5. Loggers:**

`django`: This is the default logger for all of Django’s internal logs. It captures `DEBUG` and higherlevel logs and sends them to both the console and the log file.

`django.request`: Logs errors related to HTTP requests (e.g., 404 errors, 500 errors). These logs go to the file handler.

`django.template`: Logs templaterelated errors and sends them to the console. Useful for debugging template issues.

`student`: This is a custom logger for your `student` app. It captures all logs (`DEBUG` and above) and sends them to the file handler (`django\_app.log`).

**Step 4: Use Logging in Your Application**

Once logging is configured, you can use it in your views, models, or any other module. Here's how to implement it in your Django app.

**1. Import the logging module:**

In any file where you want to log messages (e.g., `student/views.py`):

*import logging*

*logger = logging.getLogger('student') # Get a logger for the 'student' app*

**2. Log messages:**

Use the logger to log messages at different levels (`DEBUG`, `INFO`, `WARNING`, `ERROR`, `CRITICAL`):

*def my\_view(request):*

*logger.debug("This is a debug message from the student app")*

*logger.info("This is an info message")*

*logger.warning("This is a warning message")*

*logger.error("This is an error message")*

*logger.critical("This is a critical message")*

*return render(request, 'student\_template.html')*

*A screenshot of a computer program

Description automatically generated*

**3. View the logs:**

Console: When you run `python manage.py runserver`, the logs will appear in the terminal based on the console handler.

Log File: All log messages from the `student` app, and general Django logs, will also be written to the file (`logs/student\_management.log`).

A screen shot of a computer

Description automatically generated

**Step 5: Adjust Logging Levels Based on the Environment**

**Development:** In development, you generally want more detailed logs (`DEBUG` level) to help with debugging.

**Production:** In production, it’s common to set a higher logging level (`WARNING` or `ERROR`) to avoid cluttering log files with unnecessary information.

You can switch the log level based on whether the application is in development or production mode:

*if DEBUG:*

*LOGGING['loggers']['django']['level'] = 'DEBUG'*

*else:*

*LOGGING['loggers']['django']['level'] = 'WARNING'*

**Step 6: Advanced Logging Features**

Emailing Errors: You can set up a handler that sends an email whenever an `ERROR` or `CRITICAL` level log occurs.

Log Rotation: Use `logging.handlers.RotatingFileHandler` to automatically rotate logs (e.g., create a new log file once the current one reaches a certain size).

Example for Email Errors:

*'handlers': {*

*'mail\_admins': {*

*'level': 'ERROR',*

*'class': 'django.utils.log.AdminEmailHandler',*

*},*

*},*

This will send an email to the addresses specified in `ADMINS` in `settings.py` when an error occurs.