What is Logging?

If an error occurs or your app decides to work strangely, your log files will come in handy. You can traverse through them and find out where exactly the application is having problems and how you can replicate those problems.

By reproducing the problem, you can dig deeper and find a reasonable solution for the errors. Something which might otherwise take several hours to detect might just take few minutes to diagnose with the presence of log files.

The Logging Module

The logging module in Python is a ready-to-use and powerful module that is designed to meet the needs of beginners as well as enterprise teams. It is used by most of the third-party Python libraries, so you can integrate your log messages with the ones from those libraries to produce a homogeneous log for your application.

Adding logging to your Python program is as easy as this:

Python

```
import logging
```

With the logging module imported, you can use something called a “logger” to log messages that you want to see. By default, there are 5 standard levels indicating the severity of events. Each has a corresponding function that can be used to log events at that level of severity.

```
import logging

logging.debug('This is a debug message')
logging.info('This is an info message')
logging.warning('This is a warning message')
logging.error('This is an error message')
logging.critical('This is a critical message')
```

The output of the above program would look like this:

```
Shell
WARNING:root:This is a warning message
ERROR:root:This is an error message
CRITICAL:root:This is a critical message
```

How Logging Works in Django

Thankfully, Django has support for logging and most of the hard work has already been done by its developers. Django comes with Python's built-in logging module to leverage system logging.

The Python logging module has four main parts:

1. Loggers
2. Handlers
3. Filters
4. Formatters

Every component is explained meticulously in the Django Official Documentation. I don't want you to be overwhelmed with its complexity, so I'll explain every single part briefly

1. Loggers

Loggers are basically the entry point of the logging system. This is what you'll actually work with as a developers.

When a message is received by the logger, the log level is compared to the log level of the logger. If it is the same or exceeds the log level of the logger, **the message is sent to the handler for further processing.** The log levels are:

- **DEBUG:** Low-level system information
- **INFO:** General system information

- **WARNING:** Minor problems related information
- **ERROR:** Major problems related information
- **CRITICAL:** Critical problems related information

2. Handlers

Handlers basically determine what happens to each message in a logger. It has log levels the same as Loggers. But, we can essentially define what way we want to handle each log level.

For example: **ERROR** log level messages can be sent in real-time to the developer, while **INFO** log levels can just be stored in a system file.
It essentially tells the system what to do with the message like writing it on the screen, a file, or to a network socket

3. Filters

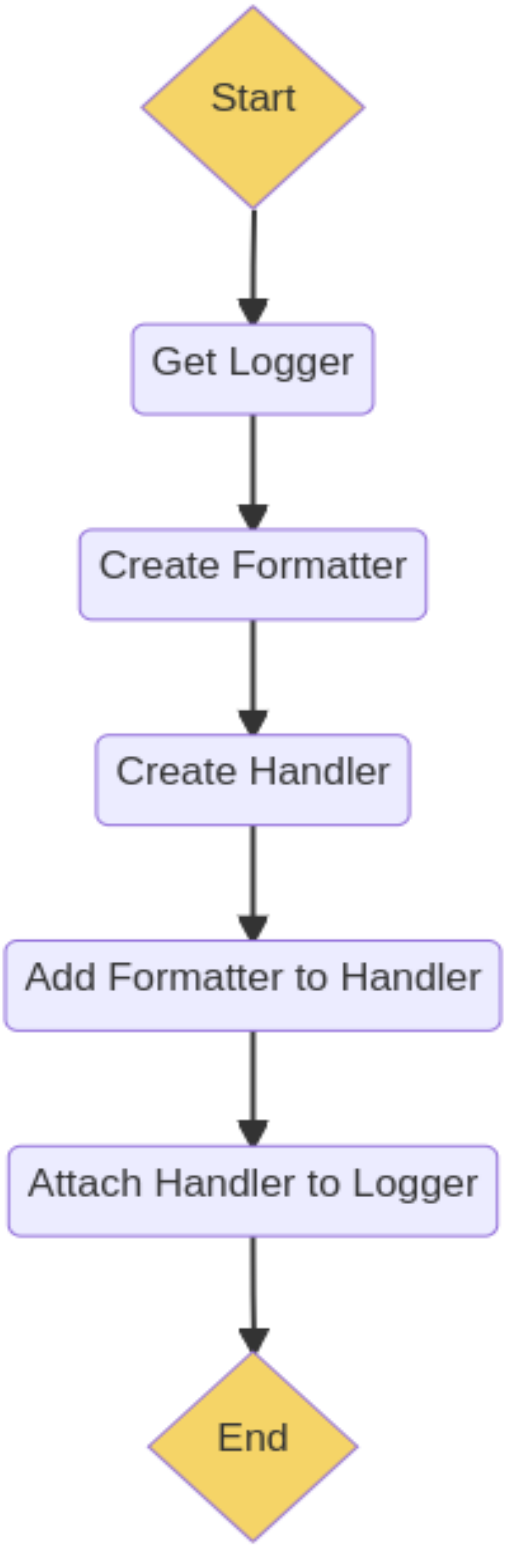
A filter can sit between a **Logger** and a **Handler**. It can be used to filter the log record. For example: in **CRITICAL** messages, you can set a filter which only allows a particular source to be processed.

4. Formatters

As the name suggests, formatters describe the format of the text which will be rendered.

Configuring logging in Django

Django's logging configuration can be done via the settings.py file of your project. By default, Django provides a minimal logging configuration that logs messages with a level of WARNING or higher to the console. To customize your logging configuration, you can update the LOGGING dictionary in the settings.py file. Here's an example of a basic logging configuration:


Start
Get Logger
Create Formatter
Create Handler
Add Formatter to Handler
Attach Handler to Logger
End

Syntax:

```
LOGGING = {
    'version': 1,
    'disable_existing_loggers': False,
    'formatters': {
    },
    'handlers': {
    },
    'loggers': {
    }
}
```

Let's now shift our focus to defining a basic logging setup in Django. We can configure several loggers, formatters, and handlers in a single location and then import them into our code as needed. The disable_existing_loggers key is set to False so that our loggers from the previous section will still work, but you may disable them if you want to restrict the use of loggers in your Django application to only the ones defined in the settings.py file (recommended).

Let's start by defining a new **Formatter**. Add the following code to the formatters dictionary shown below:

settings.py

```
'formatters': {
    'base': {
        'format': '{name} at {asctime} ({levelname}) :: {message}',
        'style': '{'
    }
```

The new **Formatter** is called **base** and it uses the same log message format in the previous section. Setting the style key to { ensures that we can reference the log message attributes using curly braces.

Similarly, a new **Handler** is created by adding it to the **handlers** dictionary within the **LOGGING** dictionary:

```
'handlers': {
    'console': {
        'class': 'logging.StreamHandler',
        'formatter': 'base'
    },
```

The console handler is a **StreamHandler** instance that logs to the console, and it uses the base Formatter created earlier to format its output.

Finally, let's define a new **Logger** under the loggers dictionary as follows

```
'loggers': {
    'horus.views.search': {
        'handlers': ['console'],
        'level': 'INFO'
    }
```

Our new logger is called **horus.views.search** and it uses the console handler defined earlier. It is also configured to only record messages with a log level of **INFO** and above.

With this configuration in place, we can access the **horus.views.search** logger in any file like this:

```
logger = logging.getLogger(__name__)
```

Add loggers into the views.py

```
import logging

logger = logging.getLogger(__name__)

def my_view(request):
    logger.info("This is an informational message.")
    logger.debug("This is a debug message.")
    logger.warning("This is a warning message.")
    logger.error("This is an error message.")
    logger.critical("This is a critical message.")
```

```
LOGGING = {
    'version': 1,
    # The version number of our log
    'disable_existing_loggers': False,
    'formatters': {
        'verbose': {
            'format': '{levelname} {asctime} {module} {message}',
            'style': '{',
        },
    },
    # django uses some of its own loggers for internal operations. In case you want to
disable them just replace the False above with true.
    # A handler for WARNING. It is basically writing the WARNING messages into a file
called WARNING.log
    'handlers': {
        'file': {
            'level': 'WARNING',
            'class': 'logging.FileHandler',
            'filename': BASE_DIR / 'warning.log',
            'formatter': 'verbose',
        },
    },
    # A logger for WARNING which has a handler called 'file'. A logger can have
multiple handler
    'loggers': {
       # notice the blank '', Usually you would put built in loggers like django or
root here based on your needs
        '': {
            'handlers': ['file'], #notice how file variable is called in handler which
has been defined above
            'level': 'WARNING',
            'propagate': True,
        },
    },
}
```

Example :

Showing the log on console and save it on file also.

```
LOGGING = {
    'version': 1,
    # The version number of our log
    'disable_existing_loggers': False,
    'formatters': {
        'verbose': {
            'format': '{levelname} {asctime} {module} {message}',
            'style': '{',
        },
    },
    # django uses some of its own loggers for internal operations. In case you want to
disable them just replace the False above with true.
    # A handler for WARNING. It is basically writing the WARNING messages into a file
called WARNING.log
    'handlers': {
        'file': {
            'level': 'WARNING',
            'class': 'logging.FileHandler',
            'filename': BASE_DIR / 'warning.log',
            'formatter': 'verbose',
        },
        'console': {
            'class': 'logging.StreamHandler',
            'formatter': 'verbose',
        },
    },
    # A logger for WARNING which has a handler called 'file'. A logger can have
multiple handler
    'loggers': {
       # notice the blank '', Usually you would put built in loggers like django or
root here based on your needs
        '': {
            'handlers': ['console','file'], #notice how file variable is called in
handler which has been defined above
            'level': 'WARNING',
            'propagate': True,
        },
    },
}
```

Simple format:

You can include the same with formater dict and add it where ever it required

```
'simple': {
            'format': '{levelname} {message}',
            'style': '{',
        },
```

```
LOGGING = {
    'version': 1,
    # The version number of our log
    'disable_existing_loggers': False,
    'formatters': {
        'verbose': {
            'format': '{levelname} {asctime} {module} {message}',
            'style': '{',
        },
        'simple': {
            'format': '{levelname} {message}',
            'style': '{',
        },
    },
    # django uses some of its own loggers for internal operations. In case you want to
disable them just replace the False above with true.
    # A handler for WARNING. It is basically writing the WARNING messages into a file
called WARNING.log
    'handlers': {
        'file': {
            'level': 'WARNING',
            'class': 'logging.FileHandler',
            'filename': BASE_DIR / 'warning.log',
            'formatter': 'verbose',
        },
        'console': {
            'class': 'logging.StreamHandler',
            'formatter': 'simple',
```

```
        },
    },
    # A logger for WARNING which has a handler called 'file'. A logger can have multiple handler
    'loggers': {
       # notice the blank '', Usually you would put built in loggers like django or root here based on your needs
        '': {
            'handlers': ['console','file'], #notice how file variable is called in handler which has been defined above
            'level': 'WARNING',
            'propagate': True,
        },
    },
}
```