Logging

See also:

- How to configure and use logging
- Django logging reference

Python programmers will often use `print()` in their code as a quick and convenient debugging tool. Using the logging framework is only a little more effort than that, but it's much more elegant and flexible. As well as being useful for debugging, logging can also provide you with more - and better structured - information about the state and health of your application.

Overview

Django uses and extends Python's builtin `logging` module to perform system logging. This module is discussed in detail in Python's own documentation; this section provides a quick overview.

The cast of players

A Python logging configuration consists of four parts:

- Loggers
- Handlers
- Filters
- Formatters

Loggers

A *logger* is the entry point into the logging system. Each logger is a named bucket to which messages can be written for processing.

A logger is configured to have a *log level*. This log level describes the severity of the messages that the logger will handle. Python defines the following log levels:

- `DEBUG` : Low level system information for debugging purposes
- `INFO` : General system information
- `WARNING` : Information describing a minor problem that has occurred.
- `ERROR` : Information describing a major problem that has occurred.
- `CRITICAL` : Information describing a critical problem that has occurred.

Each message that is written to the logger is a *Log Record*. Each log record also has a *log level* indicating the severity of that specific message. A log record can also contain useful metadata that describes the event that is being logged. This can include details such as a stack trace or an error code.

When a message is given to the logger, the log level of the message is compared to the log level of the logger. If the log level of the message meets or exceeds the log level of the logger itself, the message will undergo further processing. If it doesn't, the message will be ignored.

Once a logger has determined that a message needs to be processed, it is passed to a *Handler*.

Handlers

The *handler* is the engine that determines what happens to each message in a logger. It describes a particular logging behavior, such as writing a message to the screen, to a file, or to a network socket.

Like loggers, handlers also have a log level. If the log level of a log record doesn't meet or exceed the level of the handler, the handler will ignore the message.

A logger can have multiple handlers, and each handler can have a different log level. In this way, it is possible to provide different forms of notification depending on the importance of a message. For example, you could install one handler that forwards `ERROR` and `CRITICAL` messages to a paging service, while a second handler logs all messages (including `ERROR` and `CRITICAL` messages) to a file for later analysis.

Filters

A *filter* is used to provide additional control over which log records are passed from logger to handler.

By default, any log message that meets log level requirements will be handled. However, by installing a filter, you can place additional criteria on the logging process. For example, you could install a filter that only allows `ERROR` messages from a particular source to be emitted.

Filters can also be used to modify the logging record prior to being emitted. For example, you could write a filter that downgrades `ERROR` log records to `WARNING` records if a particular set of criteria are met.

Filters can be installed on loggers or on handlers; multiple filters can be used in a chain to perform multiple filtering actions.

Formatters

Ultimately, a log record needs to be rendered as text. *Formatters* describe the exact format of that text. A formatter usually consists of a Python formatting string containing LogRecord attributes; however, you can also write custom formatters to implement specific formatting behavior.

Security implications

The logging system handles potentially sensitive information. For example, the log record may contain information about a web request or a stack trace, while some of the data you collect in your own loggers may also have security implications. You need to be sure you know:

- what information is collected
- where it will subsequently be stored
- how it will be transferred
- who might have access to it.

To help control the collection of sensitive information, you can explicitly designate certain sensitive information to be filtered out of error reports – read more about how to filter error reports.

`AdminEmailHandler`

The built-in `AdminEmailHandler` deserves a mention in the context of security. If its `include_html` option is enabled, the email message it sends will contain a full traceback, with names and values of local variables at each level of the stack, plus the values of your Django settings (in other words, the same level of detail that is exposed in a web page when `DEBUG` is `True`).

It's generally not considered a good idea to send such potentially sensitive information over email. Consider instead using one of the many third-party services to which detailed logs can be sent to get the best of multiple worlds – the rich information of full tracebacks, clear management of who is notified and has access to the information, and so on.

Configuring logging

Python's logging library provides several techniques to configure logging, ranging from a programmatic interface to configuration files. By default, Django uses the dictConfig format.

In order to configure logging, you use `LOGGING` to define a dictionary of logging settings. These settings describe the loggers, handlers, filters and formatters that you want in your logging setup, and the log levels and other properties that you want those components to have.

By default, the `LOGGING` setting is merged with Django's default logging configuration using the following scheme.

If the `disable_existing_loggers` key in the `LOGGING` dictConfig is set to `True` (which is the `dictConfig` default if the key is missing) then all loggers from the default configuration will be disabled. Disabled loggers are not the same as removed; the logger will still exist, but will silently discard anything logged to it, not even propagating entries to a parent logger. Thus you should be very careful using `'disable_existing_loggers': True`; it's probably not what you want. Instead, you can set `disable_existing_loggers` to `False` and redefine some or all of the default loggers; or you can set `LOGGING_CONFIG` to `None` and handle logging config yourself.

Logging is configured as part of the general Django `setup()` function. Therefore, you can be certain that loggers are always ready for use in your project code.

Examples

The full documentation for dictConfig format is the best source of information about logging configuration dictionaries. However, to give you a taste of what is possible, here are several examples.

To begin, here's a small configuration that will allow you to output all log messages to the console:

`settings.py`

```
import os

LOGGING = {
    "version": 1,
    "disable_existing_loggers": False,
    "handlers": {
        "console": {
            "class": "logging.StreamHandler",
        },
    },
    "root": {
        "handlers": ["console"],
        "level": "WARNING",
    },
}
```

This configures the parent `root` logger to send messages with the `WARNING` level and higher to the console handler. By adjusting the level to `INFO` or `DEBUG` you can display more messages. This may be useful during development.

Next we can add more fine-grained logging. Here's an example of how to make the logging system print more messages from just the django named logger:

settings.py

```
import os

LOGGING = {
    "version": 1,
    "disable_existing_loggers": False,
    "handlers": {
        "console": {
            "class": "logging.StreamHandler",
        },
    },
    "root": {
        "handlers": ["console"],
        "level": "WARNING",
    },
    "loggers": {
        "django": {
            "handlers": ["console"],
            "level": os.getenv("DJANGO_LOG_LEVEL", "INFO"),
            "propagate": False,
        },
    },
}
```

By default, this config sends messages from the `django` logger of level `INFO` or higher to the console. This is the same level as Django's default logging config, except that the default config only displays log records when `DEBUG=True` . Django does not log many such `INFO` level messages. With this config, however, you can also set the environment variable `DJANGO_LOG_LEVEL=DEBUG` to see all of Django's debug logging which is very verbose as it includes all database queries.

You don't have to log to the console. Here's a configuration which writes all logging from the django named logger to a local file:

settings.py

```
LOGGING = {
    "version": 1,
    "disable_existing_loggers": False,
    "handlers": {
        "file": {
            "level": "DEBUG",
            "class": "logging.FileHandler",
            "filename": "/path/to/django/debug.log",
        },
    },
    "loggers": {
        "django": {
            "handlers": ["file"],
            "level": "DEBUG",
            "propagate": True,
        },
    },
}
```

If you use this example, be sure to change the `'filename'` path to a location that's writable by the user that's running the Django application.

Finally, here's an example of a fairly complex logging setup:

settings.py

```
LOGGING = {
    "version": 1,
    "disable_existing_loggers": False,
    "formatters": {
        "verbose": {
            "format": "{levelname} {asctime} {module} {process:d} {thread:d} {message}",
            "style": "{",
        },
        "simple": {
            "format": "{levelname} {message}",
            "style": "{",
        },
```

```
    },
    "filters": {
        "special": {
            "()": "project.logging.SpecialFilter",
            "foo": "bar",
        },
        "require_debug_true": {
            "()": "django.utils.log.RequireDebugTrue",
        },
    },
    "handlers": {
        "console": {
            "level": "INFO",
            "filters": ["require_debug_true"],
            "class": "logging.StreamHandler",
            "formatter": "simple",
        },
        "mail_admins": {
            "level": "ERROR",
            "class": "django.utils.log.AdminEmailHandler",
            "filters": ["special"],
        },
    },
    "loggers": {
        "django": {
            "handlers": ["console"],
            "propagate": True,
        },
        "django.request": {
            "handlers": ["mail_admins"],
            "level": "ERROR",
            "propagate": False,
        },
        "myproject.custom": {
            "handlers": ["console", "mail_admins"],
            "level": "INFO",
            "filters": ["special"],
        },
    },
}
```

This logging configuration does the following things:

- Identifies the configuration as being in 'dictConfig version 1' format. At present, this is the only dictConfig format version.
- Defines two formatters:
  - `simple`, that outputs the log level name (e.g., `DEBUG`) and the log message.

    The `format` string is a normal Python formatting string describing the details that are to be output on each logging line. The full list of detail that can be output can be found in Formatter Objects.
  - `verbose`, that outputs the log level name, the log message, plus the time, process, thread and module that generate the log message.
- Defines two filters:
  - `project.logging.SpecialFilter`, using the alias `special`. If this filter required additional arguments, they can be provided as additional keys in the filter configuration dictionary. In this case, the argument `foo` will be given a value of `bar` when instantiating `SpecialFilter`.
  - `django.utils.log.RequireDebugTrue`, which passes on records when DEBUG is `True`.
- Defines two handlers:
  - `console`, a StreamHandler, which prints any `INFO` (or higher) message to `sys.stderr`. This handler uses the `simple` output format.
  - `mail_admins`, an AdminEmailHandler, which emails any `ERROR` (or higher) message to the site ADMINS. This handler uses the `special` filter.
- Configures three loggers:
  - `django`, which passes all messages to the `console` handler.
  - `django.request`, which passes all `ERROR` messages to the `mail_admins` handler. In addition, this logger is marked to *not* propagate messages. This means that log messages written to `django.request` will not be handled by the `django` logger.
  - `myproject.custom`, which passes all messages at `INFO` or higher that also pass the `special` filter to two handlers – the `console`, and `mail_admins`. This means that all `INFO` level messages (or higher) will be printed to the console; `ERROR` and `CRITICAL` messages will also be output via email.

Custom logging configuration

If you don't want to use Python's dictConfig format to configure your logger, you can specify your own configuration scheme.

The `LOGGING_CONFIG` setting defines the callable that will be used to configure Django's loggers. By default, it points at Python's `logging.config.dictConfig()` function. However, if you want to use a different configuration process, you can use any other callable that takes a single argument. The contents of `LOGGING` will be provided as the value of that argument when logging is configured.

Disabling logging configuration

If you don't want to configure logging at all (or you want to manually configure logging using your own approach), you can set `LOGGING_CONFIG` to `None`. This will disable the configuration process for Django's default logging.

Setting `LOGGING_CONFIG` to `None` only means that the automatic configuration process is disabled, not logging itself. If you disable the configuration process, Django will still make logging calls, falling back to whatever default logging behavior is defined.

Here's an example that disables Django's logging configuration and then manually configures logging:

settings.py

```
LOGGING_CONFIG = None

import logging.config

logging.config.dictConfig(...)
```

Note that the default configuration process only calls `LOGGING_CONFIG` once settings are fully-loaded. In contrast, manually configuring the logging in your settings file will load your logging config immediately. As such, your logging config must appear *after* any settings on which it depends.