Logging ¶

A quick logging primer ¶

Django uses Python's builtin **`logging`** module to perform system logging. The usage of this module is discussed in detail in Python's own documentation. However, if you've never used Python's logging framework (or even if you have), here's a quick primer.

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

Browse

- Prev: Time zones

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.

Using logging ¶

Once you have configured your loggers, handlers, filters and formatters, you need to place logging calls into your code. Using the logging framework is very simple. Here's an example:

```
# import the logging library
import logging

# Get an instance of a logger
logger = logging.getLogger(__name__)

def my_view(request, arg1, arg):
    ...
    if bad_mojo:
        # Log an error message
        logger.error('Something went wrong!')
```

And that's it! Every time the **bad_mojo** condition is activated, an error log record will be written.

Naming loggers ¶

The call to **logging.getLogger()** obtains (creating, if necessary) an instance of a logger. The logger instance is identified by a name. This name is used to identify the logger for configuration purposes.

By convention, the logger name is usually **__name__**, the name of the python module that contains the logger. This allows you to filter and handle logging calls on a per-module basis. However, if you have some other way of organizing your logging messages, you can provide any dot-separated name to identify your logger:

```
# Get an instance of a specific named logger
logger = logging.getLogger('project.interesting.stuff')
```

The dotted paths of logger names define a hierarchy. The **project.interesting** logger is considered to be a parent of the **project.interesting.stuff** logger; the **project** logger is a parent of the **project.interesting** logger.

- Table of contents
- General Index
- Python Module Index

You are here:

- Django 1.11 documentation
  - Using Django
    - Logging

Getting help

FAQ
Try the FAQ — it's got answers to many common questions.

Index, Module Index, or Table of Contents
Handy when looking for specific information.

django-users mailing list
Search for information in the archives of the django-users mailing list, or post a question.

#django IRC channel
Ask a question in the #django IRC channel, or search the IRC logs to see if it's been asked before.

Django Discord Server
Join the Django Discord Community.

Official Django Forum
Join the community on the Django Forum.

Ticket tracker
Report bugs with Django or Django documentation in our ticket tracker.

Download:

Offline (Django 1.11): HTML | PDF | ePub
Provided by Read the Docs.

and capture all logging calls in the subtree of loggers. A logging handler defined in the **`project`** namespace will catch all logging messages issued on the **`project.interesting`** and **`project.interesting.stuff`** loggers.

This propagation can be controlled on a per-logger basis. If you don't want a particular logger to propagate to its parents, you can turn off this behavior.

Making logging calls ¶

The logger instance contains an entry method for each of the default log levels:

- **`logger.debug()`**
- **`logger.info()`**
- **`logger.warning()`**
- **`logger.error()`**
- **`logger.critical()`**

There are two other logging calls available:

- **`logger.log()`**: Manually emits a logging message with a specific log level.
- **`logger.exception()`**: Creates an **`ERROR`** level logging message wrapping the current exception stack frame.

Configuring logging ¶

Of course, it isn't enough to just put logging calls into your code. You also need to configure the loggers, handlers, filters and formatters to ensure that logging output is output in a useful way.

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 describes 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 default) 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.

First, here's a simple configuration which writes all logging from the django logger to a local file:

```
LOGGING = {
    'version': 1,
    'disable_existing_loggers': False,
    'handlers': {
        'file': {
            'level': 'DEBUG',
```

```
        },
    },
    '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.

Second, here's an example of how to make the logging system print Django's logging to the console. It may be useful during local development.

By default, this config only sends messages of level **`INFO`** or higher to the console (same as Django's default logging config, except that the default only displays log records when **`DEBUG=True`**). Django does not log many such 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:

```
import os

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

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

```
LOGGING = {
    'version': 1,
    'disable_existing_loggers': False,
    'formatters': {
        'verbose': {
            'format': '%(levelname)s %(asctime)s %(module)s %
(process)d %(thread)d %(message)s'
        },
        'simple': {
            'format': '%(levelname)s %(message)s'
        },
    },
    'filters': {
        'special': {
            '()': 'project.logging.SpecialFilter',
            'foo': 'bar',
        },
        'require_debug_true': {
            '()': 'django.utils.log.RequireDebugTrue',
        },
    },
    'handlers': {
        'console': {
```

```
            '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 just 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 will print any **`INFO`** (or higher) message to stderr. This handler uses the **`simple`** output format.
  - **`mail_admins`**, an AdminEmailHandler, which will email 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

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. 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(...)
```

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.

Django's logging extensions ¶

Django provides a number of utilities to handle the unique requirements of logging in Web server environment.

Loggers ¶

Django provides several built-in loggers.

django ¶

The catch-all logger for messages in the **django** hierarchy. No messages are posted using this name but instead using one of the loggers below.

django.request ¶

Log messages related to the handling of requests. 5XX responses are raised as **ERROR** messages; 4XX responses are raised as **WARNING** messages.

Messages to this logger have the following extra context:

- **status_code**: The HTTP response code associated with the request.
- **request**: The request object that generated the logging message.

django.server ¶

New in Django 1.10.

logged as **`WARNING`** messages, and everything else is logged as **`INFO`**.

Messages to this logger have the following extra context:

- **`status_code`**: The HTTP response code associated with the request.
- **`request`**: The request object that generated the logging message.

`django.template` ¶

Log messages related to the rendering of templates.

- Missing context variables are logged as **`DEBUG`** messages.
- Uncaught exceptions raised during the rendering of an **`{% include %}`** are logged as **`WARNING`** messages when debug mode is off (helpful since **`{% include %}`** silences the exception and returns an empty string in that case).

`django.db.backends` ¶

Messages relating to the interaction of code with the database. For example, every application-level SQL statement executed by a request is logged at the **`DEBUG`** level to this logger.

Messages to this logger have the following extra context:

- **`duration`**: The time taken to execute the SQL statement.
- **`sql`**: The SQL statement that was executed.
- **`params`**: The parameters that were used in the SQL call.

For performance reasons, SQL logging is only enabled when **`settings.DEBUG`** is set to **`True`**, regardless of the logging level or handlers that are installed.

This logging does not include framework-level initialization (e.g. **`SET TIMEZONE`**) or transaction management queries (e.g. **`BEGIN`**, **`COMMIT`**, and **`ROLLBACK`**). Turn on query logging in your database if you wish to view all database queries.

`django.security.*` ¶

The security loggers will receive messages on any occurrence of **`SuspiciousOperation`** and other security-related errors. There is a sub-logger for each subtype of security error, including all **`SuspiciousOperation`**s. The level of the log event depends on where the exception is handled. Most occurrences are logged as a warning, while any **`SuspiciousOperation`** that reaches the WSGI handler will be logged as an error. For example, when an HTTP **`Host`** header is included in a request from a client that does not match **`ALLOWED_HOSTS`**, Django will return a 400 response, and an error message will be logged to the **`django.security.DisallowedHost`** logger.

These log events will reach the **`django`** logger by default, which mails error events to admins when **`DEBUG=False`**. Requests resulting in a 400 response due to a **`SuspiciousOperation`** will not be logged to the **`django.request`** logger, but only to the **`django.security`** logger.

To silence a particular type of **`SuspiciousOperation`**, you can override that specific logger following this example:

```
'handlers': {
    'null': {
        'class': 'logging.NullHandler',
    },
},
'loggers': {
    'django.security.DisallowedHost': {
        'handlers': ['null'],
        'propagate': False,
    },
},
```

- **`django.security.csrf`**: For CSRF failures.

`django.db.backends.schema` ¶

Logs the SQL queries that are executed during schema changes to the database by the migrations framework. Note that it won't log the queries executed by **`RunPython`**. Messages to this logger have **`params`** and **`sql`** in their extra context (but unlike **`django.db.backends`**, not duration). The values have the same meaning as explained in django.db.backends.

New in Django 1.10:

The **`extra`** context was added.

Handlers ¶

Django provides one log handler in addition to those provided by the Python logging module.

class **`AdminEmailHandler`**(*include_html=False*, *email_backend=None*)[source] ¶

This handler sends an email to the site admins for each log message it receives.

If the log record contains a **`request`** attribute, the full details of the request will be included in the email. The email subject will include the phrase "internal IP" if the client's IP address is in the **`INTERNAL_IPS`** setting; if not, it will include "EXTERNAL IP".

If the log record contains stack trace information, that stack trace will be included in the email.

The **`include_html`** argument of **`AdminEmailHandler`** is used to control whether the traceback email includes an HTML attachment containing the full content of the debug Web page that would have been produced if **`DEBUG`** were **`True`**. To set this value in your configuration, include it in the handler definition for **`django.utils.log.AdminEmailHandler`**, like this:

```
'handlers': {
    'mail_admins': {
        'level': 'ERROR',
        'class': 'django.utils.log.AdminEmailHandler',
        'include_html': True,
    }
},
```

Note that this HTML version of the email contains a full traceback, with names and values of local variables at each level of the stack, plus the values of your Django settings. This information is potentially very sensitive, and you may not want to send it over email. Consider using something such as Sentry to get the best of both worlds – the rich information of full tracebacks plus the security of *not* sending the information over email. You may also explicitly designate certain sensitive information to be filtered out of error reports – learn more on Filtering error reports.

By setting the **`email_backend`** argument of **`AdminEmailHandler`**, the email backend that is being used by the handler can be overridden, like this:

```
'handlers': {
    'mail_admins': {
        'level': 'ERROR',
        'class': 'django.utils.log.AdminEmailHandler',
        'email_backend':
'django.core.mail.backends.filebased.EmailBackend',
    }
},
```

By default, an instance of the email backend specified in **`EMAIL_BACKEND`** will be used.

Sends emails to admin users. To customize this behavior, you can subclass the **AdminEmailHandler** class and override this method.

Filters ¶

Django provides two log filters in addition to those provided by the Python logging module.

class CallbackFilter(*callback*)[source] ¶

This filter accepts a callback function (which should accept a single argument, the record to be logged), and calls it for each record that passes through the filter. Handling of that record will not proceed if the callback returns False.

For instance, to filter out **UnreadablePostError** (raised when a user cancels an upload) from the admin emails, you would create a filter function:

```
from django.http import UnreadablePostError

def skip_unreadable_post(record):
    if record.exc_info:
        exc_type, exc_value = record.exc_info[:2]
        if isinstance(exc_value, UnreadablePostError):
            return False
    return True
```

and then add it to your logging config:

```
'filters': {
    'skip_unreadable_posts': {
        '()': 'django.utils.log.CallbackFilter',
        'callback': skip_unreadable_post,
    }
},
'handlers': {
    'mail_admins': {
        'level': 'ERROR',
        'filters': ['skip_unreadable_posts'],
        'class': 'django.utils.log.AdminEmailHandler'
    }
},
```

class RequireDebugFalse[source] ¶

This filter will only pass on records when settings.DEBUG is False.

This filter is used as follows in the default **LOGGING** configuration to ensure that the **AdminEmailHandler** only sends error emails to admins when **DEBUG** is **False**:

```
'filters': {
    'require_debug_false': {
        '()': 'django.utils.log.RequireDebugFalse',
    }
},
'handlers': {
    'mail_admins': {
        'level': 'ERROR',
        'filters': ['require_debug_false'],
        'class': 'django.utils.log.AdminEmailHandler'
    }
},
```

class RequireDebugTrue[source] ¶

Django's default logging configuration ¶

By default, Django configures the following logging:

When **DEBUG** is **True**:

- The **django** logger sends messages in the **django** hierarchy (except **django.server**) at the **INFO** level or higher to the console.

When **DEBUG** is **False**:

- The **django** logger sends messages in the **django** hierarchy (except **django.server**) with **ERROR** or **CRITICAL** level to **AdminEmailHandler**.

Independent of the value of **DEBUG**:

- The django.server logger sends messages at the **INFO** level or higher to the console.

See also Configuring logging to learn how you can complement or replace this default logging configuration.

Time zones

Pagination