Learning Objectives

- **Identify the different logging levels**
- **Add a logger to your Django project**
- **Use a formatter to provide more context to logging messages**
- **Identify import logging concepts that go beyond 12 Factor Apps**

Clone Blango Repo

Clone Blango Repo

Before we continue, you need to clone the `blango` repo so you have all of your code. You will need the SSH information for your repo.

In the Terminal

- Clone the repo. Your command should look something like this:

```
git clone git@github.com:<your_github_username>/blango.git
```

- You should see a `blango` directory appear in the file tree.

You are now ready for the next assignment.

Logging Intro

Logging Intro

There's no getting past it: logging is a complicated topic. We won't be able to cover everything there is to know about logging here, but we'll give you the fundamentals. Since we're targeting the 12-Factor method of logging, it simplifies the process since we only need to log to stdout/the console.

Logging consists of four parts:

- Loggers
- Handlers
- Filters
- Formatters

Loggers

A logger is the entry point to the logging system. It's like a named bucket that messages get written to. In Python you get a logger by name. The first time you get a logger by name it's created, and subsequent requests for a logger with that name will give you the same one. A logger also has a *log level*, which describes the severity of messages it will handle.

The log levels to choose from are:

- `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.

After you have a logger, you can write a *Log Record* to it. A log record consists of a message, a log level, and optionally some metadata (like a stack trace). A logger will only handle messages at its log level or higher. For example, a logger at log level `WARNING` will handle `WARNING`, `ERROR` and `CRITICAL` messages, but not `DEBUG` or `INFO`. Now let's look at what handling means.

Handlers

After a logger receives a log record that exceeds its log level, it's passed on to a *Handler*. The handler decides where the log record will go, such as to the console, to a file, or somewhere else. The handler also has its own log level, so it will only handle log records which match or exceed that.

A logger can have multiple handlers, so you could have a handler that messages admins for `ERROR` and `CRITICAL` alerts, while logging all messages to console for later analysis.

Filters

A filter controls how data log records are passed from logger to handler. A filter is not required for a handler/logger, but you'd want one if you need extra customization. You can investigate each log record and choose to drop it, or forward it on to a handler. You can even make changes to a log record's log level before passing it on.

Filters can be installed on loggers or handlers, and multiple filters can be chained together.

Formatters

Before a log message is output, it is formatted, by a formatter. The formatter might just output the message verbatim, or it could prepend the time and date, and process information. It could also truncate the message, or anything else you choose.

Now let's look at logging in the context of real Django code.

How To Log

Logging in Python is done using the `logging` module. Although, the only function we're interested in is `getLogger()`, which takes the name of the logger as an argument, and it returns a logger instance. You can use any name for the logger but the convention is to pass in the special variable `__name__`, which is the name of the current module. For example, the value of `__name__` in the `blog/views.py` file is `blog.views`. In `blango/urls.py` it's `blango.urls`. Import the `logging` module and instantiate a `logging` variable in the IDE to the left:

```
import logging

logger = logging.getLogger(__name__)
```

Using `__name__` provides an automatic natural hierarchy of log configuration: loggers look for their configuration by traversing up their module path. Setting a configuration for `blog` also applies to `blog.views`, and `blog.models`, etc, unless it's overridden by a more specific config. If we had two configurations, `blog` and `blog.models`, then `blog` would apply to the `blog.view`, `blog.admin`, `blog.forms`, etc, loggers; `blog.models` would apply just to the `blog.models` logger. If no configuration is specified for a module, then Python falls back to using the configuration for `root`.

The logger instance provides a number of methods for logging, one for each of the log levels.

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

Each of these has the same signature (so we'll just look at `debug`):

```
logger.debug(msg, *args, **kwargs)
```

`msg` is a string with old-style format parameters (like `%s` for strings or `%d` for integers, or named parameters like `%(value)s`). `msg` is then interpolated with the values in `*args` and `**kwargs`.

challenge

Try it out:

Experiment with the different logging methods:

```
logger.debug("This is a debug message")
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")
```

▼ **What happened to two of the messages?**
The default security level is `WARNING`, so the logging module will not log messages with `DEBUG` and `INFO`.

For performance reasons, pre-interpolated strings shouldn't be passed to the logging function.

For example, these two lines would produce the same log output:

```
logger.debug("Created user %s with email %s" % (username,
        email))
logger.debug("Created user %s with email %s", username, email)
```

However, in the first usage, the string is formatted and then sent to the logging function. If the logging level of the logger is not `DEBUG` then the message will be discarded.

In the second call, if the logger level is not `DEBUG` then the message is discarded before it's interpolated, thus preventing the (admittedly small) overhead of this operation.

`**kwargs` can contain other values for adding metadata to the log record, but these are beyond the scope of this course. You can read about them at the Python logging documentation

There are two other methods for writing logs:

- `logger.log()`, which adds a `level` argument, so that they level can be provided dynamically. This call is the equivalent of our previous example:

```
logger.log(logging.DEBUG, "Created user %s with email %s",
        username, email)
```

- `logger.exception()`, which logs the current exception and stack trace, if called inside an `except` handler. It logs at log level `ERROR`, and automatically determines the current exception from the context in which it's called. You'd use it like this:

```
try:
    # some code that might raise an exception
    raise_exception()
except ValueError:
    logger.exception("An exception occured")
```

And the output of this script contains the message that's logged, as well as the stack trace:

```
An exception occured
Traceback (most recent call last):
  File "/Users/ben/logger_test.py", line 13, in main
    raise_exception()
  File "/Users/ben/logger_test.py", line 8, in raise_exception
    raise ValueError("A test exception.")
ValueError: A test exception.
```

Note once again how the exception doesn't need to be passed to the `logger.exception()` call.

challenge

Try it out:

- Use the `logger.log` method to log a message.

```
username = "example_username"
email = "example_email@mail.com"

logger.log(logging.WARNING, "Created user %s with email %s",
        username, email)
```

- Use `logger.exception` to log an exception and stack trace.

```
try:
    answer = 9 / 0
    print(f"The answer is: {answer}")
    raise_exception()
except ZeroDivisionError:
    logger.exception("A divide by zero exception occured")
```

The convention when creating a logger is to define it as a module-level variable called `logger`, at the top of a Python file. This way it's easily usable anywhere throughout the file.

Adding Logging to Django

Adding Logging to a Django Project

Now that you know about log levels and how to log data, adding logging to a Django project is simple.

1. Import the `logging` module at the top of your Python file.
2. Create a module-level variable, before any of your functions or classes, by adding the line:

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

3. Anywhere throughout the file where you want to log, add logger calls, `logger.debug()`, `logger.info()`, etc. You can add as many was you want, and should use the logging configuration to turn them on or off.

Let's try this out in Blango. Our application isn't too complex, so we'll just add some logging to the `view.py` file.

Start by following steps 1 and 2 above. For step 3, we're going to log two things.

- After retrieving the `Post`s from the database in the `index` view, log how many they are at `DEBUG` level:

```
logger.debug("Got %d posts", len(posts))
```

- In the `post_detail` view, log a message when a `Comment` is created. This should be added just after the `Comment` is saved.

```
logger.info(
    "Created comment on Post %d for user %s", post.pk,
        request.user
)
```

Save your changes, and start the dev server. Leave a comment or create a post while watching the console where your Django dev server is running.

View Blog

You should see... that nothing is logged! Before Django will output our logs, we need to configure how it logs (i.e. the handlers, formatters and filters) in `settings.py`.

Django Logging Configuration Part 1

Django Logging Configuration Part 1

Django reads logging settings from the `LOGGING` setting. The settings are configured by a dictionary, which follows a standard Python schema for dictConfig. The dictionary has these keys:

- `version`: an integer describing the version of the schema. Currently must be `1` but is in place for backwards compatibility if the schema changes in the future. This is the only key that is required to be set, all the following are optional.
- `formatters`: A dictionary mapping a formatter's ID (string) to a dictionary containing its configuration.
- `filters`: A dictionary mapping a filter's ID (string) to a dictionary containing its configuration.
- `handlers`: A dictionary mapping a handler's ID (string) to a dictionary containing its configuration.
- `loggers`: A dictionary mapping a loggers's ID (string) to a dictionary containing its configuration.
- `root`: Contains a dictionary with configuration for the root (default) logger. That is, instead of configuring the root logger in `LOGGING["loggers"]["root"]` it's in `LOGGING["root"]`
- `incremental`: A boolean, `False` (the default) means that this config dictionary replaces any existing config. When set to `True` the configuration is "incrementally applied" to existing configuration. This is an advanced option which we won't be using, but if you're curious about the specifics and use cases then check the incremental configuration documentation.
- `disable_existing_loggers`: Disable any existing non-root loggers. Defaults to `True`.

The keys (ID strings) for `formatters`, `filters` and `handlers` are all arbitrary. They are used in the logger configuration as references. For example, a formatter could have the ID `verbose` (to indicate a verbose output), and then the `handler` can use the string `verbose` to refer to it. Similarly, a handler could have the ID `console` (indicating that it will output to the console), and the `logger` can use the string `console` to refer to this. We'll see how to define these soon.

The keys the `loggers` match the logger's name to which they are applied. For example, `blog` to apply configuration to our `blog` app, or `blog.views` for just the `blog.views` module (`blog/views.py` file).

Let's look at a real example, then go through how it works. You will now update Blango to log your messages, as well as experiment with some other log settings. Start by adding the basic `LOGGING` configuration to your `Dev` settings class.

```
LOGGING = {
    "version": 1,
    "disable_existing_loggers": False,
    "handlers": {
        "console": {"class": "logging.StreamHandler", "stream":
        "ext://sys.stdout"},
    },
    "root": {
        "handlers": ["console"],
        "level": "DEBUG",
    }
}
```

This configuration sets up one handler, with the ID `console`. While the ID is arbitrary and doesn't determine how the handler works, it's good to pick one that makes sense for how the handler behaves. In this case, the handler will log to the console. The handler's configuration dictionary requires setting its `class`, which is the class of the handler to instantiate. We can also provide values for the keys:

- `level`: (the log level to set for the handler, as a string like `"DEBUG"`, `"WARNING"`, etc).
- `formatter`: The ID of the formatter for the handler to use.
- `filters`: A list of filter IDs for the hander to use.

Any other keys are passed to the class as `**kwargs` to instantiate it. By default, the `StreamHandler` class logs to STDERR, so we change it to stdout by setting its `stream` argument to the URL `ext://sys.stdout`.

▼ **StreamHandler**

Since we're following the 12 Factor method of logging, we'll just be using the `StreamHandler`, but a list of all the Python handlers is available at the logging handlers documentation. For example, we could set up logging to a file like this:

```
"handlers": {
  "file": {"class": "logging.FileHandler", "filename":
        "/var/log/blango.log"},
}
```

And if you're feeling adventurous, you can even write your own handler!

Returning back to our logging config, the next key we've set is `root`, which sets the default/root logging options. The list of `handlers` is set to just `console`, and we set the `level` to `DEBUG`. We can also provide the keys:

- `propagate`: `True` to propagate messages to handlers higher up in the hierarchy, or `False` to not. Defaults to `True`.
- `filters`: A list of filters IDs of filters to pass the log record through.

Start the Django development server and watch the console in which it's running. Then, view the post list page in your browser. You should see output in the console like:

View Blog

```
Got 7 posts
```

Then, try commenting on a Post. You'll see a message like:

```
Created comment on Post 6 for user ben
```

challenge

Try it out:

You can try experimenting with different log levels to see what causes the log message to be shown or not. If you set the `level` of the handler or `root` to `WARNING` or higher, you won't see any messages. Setting it to `INFO` will show just the `INFO` level message and not the `DEBUG` one.

When you're finished experimenting, change the level back to `DEBUG`.

▼ **Original Code**

After changing the level back to `DEBUG`, your code should look like this:

```
LOGGING = {
  "version": 1,
  "disable_existing_loggers": False,
  "handlers": {
      "console": {"class": "logging.StreamHandler",
        "stream": "ext://sys.stdout"},
  },
  "root": {
      "handlers": ["console"],
      "level": "DEBUG",
  }
}
```

Django Logging Configuration Part 2

Django Logging Configuration Part 2

Since we're following the 12 Factor app we want to log all log records to stdout and allow the operating system to configure how they're routed later. However, if we just have the message and no other metadata it's hard for the OS to perform any meaningful routing. To allow routing by showing the log level, as well as show more useful information like the time of the message and the process/thread ID, we should use a formatter.

The configuration dictionary for a formatter can have three keys, all are optional:

- `fmt`: A string for the message format. Defaults to `"%(message)s"`.
- `datefmt`: The format to use for the date/time of the log record. Defaults to `"%Y-%m-%d %H:%M:%S,uuu"`.
- `style`: This is a string indicating the style of the tokens in the `fmt` string. Either `"{"` for new/template style string tokens (e.g. `"{message}"`) or `"%s"` for old style interpolation. `"%s"` is the default.

Here's an example of `LOGGING` configuration with a verbose formatter (again, the name is arbitrary, but it's called `verbose` because it's quite verbose), which is set on the `console` handler.

```
LOGGING = {
    "version": 1,
    "disable_existing_loggers": False,
    "formatters": {
        "verbose": {
            "format": "{levelname} {asctime} {module}
        {process:d} {thread:d} {message}",
            "style": "{",
        },
    },
    "handlers": {
        "console": {
            "class": "logging.StreamHandler",
            "stream": "ext://sys.stdout",
            "formatter": "verbose",
        },
    },
    "root": {
        "handlers": ["console"],
        "level": "DEBUG",
    },
}
```

The formatter will output the log level, time of the message (`asctime`), the name of the module that generated the message, the process ID, thread ID, and finally the message.

Update your `LOGGING` configuration as per the example above, to add a `verbose` formatter, and have the `console` handler use it. Try performing one of the actions that produces a log message, and notice how the format of it has changed in the console:

View Blog

```
DEBUG 2021-07-24 07:25:20,176 views 74341 123145456508928 Got 7
        posts
```

Non 12-Factor Considerations

If you've already used Django in production, you might have noticed that admin users will receive an email alert when an exception occurs (provided you've configured Django's SMTP settings correctly too).

▼ **Emailing Admins**

By “admin users” in the sentence above, we don’t mean users with access to the Django admin site. We mean users listed in the `ADMINS` setting. This is a list of two-element tuples containing the admin name and email address. For example:

```
ADMINS = [("Ben Shaw", "ben@example.com"), ("Leo Lucio",
        "leo@example.com")]
```

And to go further aside, how would the above setting be read from an environment variable with Django Configurations? You should use a `SingleNestedTupleValue` and set the environment variable like this:

```
DJANGO_ADMINS="Ben Shaw,ben@example.com;Leo
        Lucio,leo@example.com"
```

Alerts-on-exception emails are triggered by the log handler `django.utils.log.AdminEmailHandler`. Even though the 12 Factor methodology suggest sending logs only to the console, it can be useful to have these admin emails too. We can add this feature back to our `LOGGING` config like this:

```
LOGGING = {
    "version": 1,
    "disable_existing_loggers": False,
    "filters": {
        "require_debug_false": {
            "()": "django.utils.log.RequireDebugFalse",
        },
    },
    "formatters": {
        "verbose": {
            "format": "{levelname} {asctime} {module}
        {process:d} {thread:d} {message}",
            "style": "{",
        },
    },
    "handlers": {
        "console": {
            "class": "logging.StreamHandler",
            "stream": "ext://sys.stdout",
            "formatter": "verbose",
        },
        "mail_admins": {
            "level": "ERROR",
            "class": "django.utils.log.AdminEmailHandler",
            "filters": ["require_debug_false"],
        },
    },
    "loggers": {
        "django.request": {
            "handlers": ["mail_admins"],
            "level": "ERROR",
            "propagate": True,
        },
    },
    "root": {
        "handlers": ["console"],
        "level": "DEBUG",
    },
}
```

We've also introduced a filter which we've given the ID `require_debug_false`. This uses the filter class `django.utils.log.RequireDebugFalse`, which is a class that only passes message through when the `DEBUG` settings is `False`. We apply this filter to `mail_admins` handler, so that error emails are only sent in production environments (otherwise they'd be inundated as we developed our Django apps). Finally we set this handler only for the `django.request`, so only when

exceptions are unhandled does it get sent. We make sure to add `"propagate": True` so that the stack traces also get logged to the console during development.

challenge

Try it out:

If you want, you can set the above `LOGGING` config into your Blango settings. You shouldn't notice a difference from the last settings, during development at least. But if you do decide to take Blango to production, it's nice to have admins receiving email alerts as per the Django default.

Pushing to GitHub

Pushing to GitHub

Before continuing, you must push your work to GitHub. In the terminal:

- Commit your changes:

```
git add .
git commit -m "Finish logging"
```

- Push to GitHub:

```
git push
```