# PARTE I: BEST PRACTICE PER IL LOGGING SU PYTHON E COME INTEGRARSI CON LA DASHBOARD DI KIBANA TRAMITE AWS KINESIS DATA FIREHOSE ED AMAZON ELASTICSEARCH SERVICE

Amazon Elasticsearch Service | Amazon Kinesis Data Firehose | Kibana | Python

beSharp | 15 Maggio 2020

---

Le applicazioni, nel momento in cui passano in produzione, perdono la capacità di fornire direttamente informazioni su cosa sta succedendo dietro le quinte del codice, diventano delle scatole nere e quindi si ha la necessità di tracciare e monitorare efficacemente il loro comportamento. **Il logging** è sicuramente la procedura più semplice e diretta per approcciare il problema. Siamo infatti in grado di istruire il programma, in fase di sviluppo, ad emettere informazioni durante il suo funzionamento, e queste saranno utili per svolgere attività di troubleshooting o semplicemente per effettuare analisi.

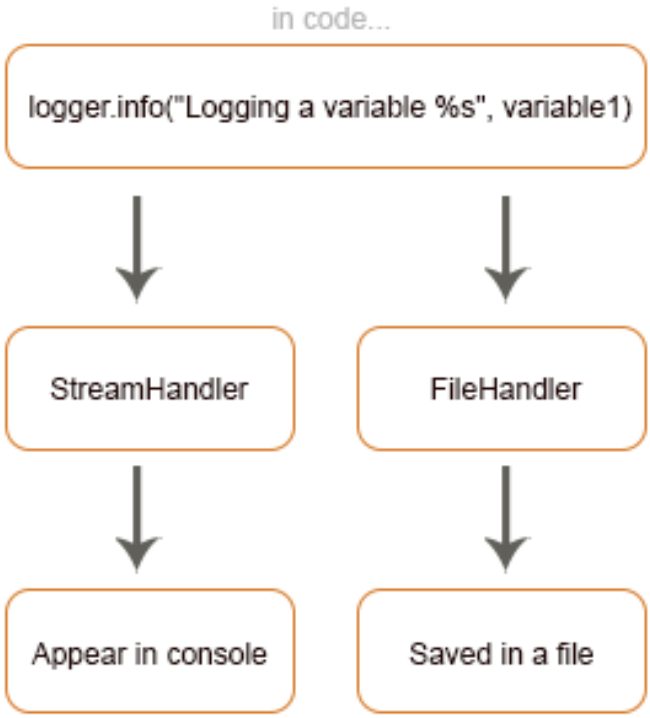
Quando si parla di **Python**, il modulo già incluso per il logging è progettato per funzionare direttamente con una fase di setup veramente minimale. Indipendentemente dal livello di esperienza come sviluppatore e del modulo di logging di Python, lo scopo di questo articolo è di essere il più esauriente possibile e di indagare a fondo le metodologie che permettono di sfruttare il logging al massimo.

Una nota a margine: questo articolo si compone di due parti. La seconda parte verterà su come approcciarsi ai servizi di AWS, utilizzando una combinazione di **AWS Kinesis Data Firehose** e **AWS ElasticSearch** per realizzare un sistema efficace di ricerca ed analisi dei log attraverso la **dashboard di Kibana** e di permetterne la copia su S3 per mantenere uno storico dei log a portata di mano.

## Le basi del modulo di logging di Python

Prima di iniziare il nostro viaggio, facciamoci una semplice domanda: **che cos'è un Logger?** In pratica si tratta di un oggetto con cui uno sviluppatore interagisce per permettere la stampa di informazioni. Sono strumenti che utilizziamo per indicare al sistema che governa il programma come e cosa si voglia loggare.

Se prendiamo una istanza generica di un logger, possiamo inviare messaggi quando vogliamo senza preoccuparci della sua implementazione sottostante. Ad esempio, se si scrive logger.info("Logging a variable %s", variable1) si va ad indicare la seguente situazione:




La Libreria standard di Python fornisce un modulo di logging che permette di iniziare ad utilizzarlo senza nessun tipo di installazione aggiuntiva. Per aderire alle **best practice** che richiedono di evitare la semplice scrittura in console di quello che vogliamo riportare, il modulo di Python per il logging offre diversi vantaggi:

- supporto per il multi-threading;
- livelli di logging;
- più flessibilità;
- separazione tra dati e formattazione.

Si può esprimere il potenziale del logging proprio perché si va a separare il contenuto del log dalla sua forma. Prendendo in esame la sua forma, è necessario considerare tre aspetti principali:

1. **Level**: il livello minimo di priorità dei messaggi di cui fare log. In ordine crescente di severità, i livelli sono: DEBUG, INFO, WARNING, ERROR e CRITICAL. Di base il livello è settato su WARNING, ciò significa che il modulo di logging di Python filtrerà tutti i messaggi impostati a DEBUG o INFO.
2. **handler**: determina dove verranno salvati i log. Se non specificato, la libreria di logging utilizzerà uno StreamHandler per inviare i messaggi al sys.stderr o, per meglio intenderci, verso la console. Gestisce anche la formattazione.
3. **format**: di base, la libreria propone la seguente formattazione: <LEVEL>:<LOGGER_NAME>: <MESSAGE>. Naturalmente è data la possibilità di modificare lo stile delle righe di log,

aggiungendo eventualmente informazioni custom, ma questa parte verrà affrontata più avanti.

**Il modulo di logging di Python** fornisce diverse modalità di configurazione dei logger. Le configurazioni consentono di combinare handler, formati e livelli per andare incontro a necessità specifiche. Cominciamo dalla configurazione più semplice, quella mediante basicConfig().

## basicConfig()

Utilizzando il metodo basicConfig() è possibile **configurare rapidamente il comportamento desiderato** del proprio logger di root. Altri approcci gestibili per progetti di grandi dimensioni includono invece l'utilizzo di configurazioni di logging basate su file o su dizionario.

Tuttavia, puntare su basicConfig() è il modo preferito per iniziare a descrivere come affrontare al meglio la configurazione del logging.

Il modulo di logging, come detto poc'anzi, è settato su WARNING come livello predefinito. Ciò significa che, in determinate situazioni, si potrebbe non avere visibilità durante la ricerca di errori o, in generale, qualora si debba condurre una root cause analysis. In base ai suoi standard, il modulo di logging invia i log direttamente alla console. Altre opzioni, possibili e raccomandate, prevedono l'uso di StreamHandler o SocketHandler per lo streaming in rete e di FileHandler per accedere direttamente al disco.

Anche se il salvataggio su file ha i suoi vantaggi, come quello di evitare potenziali errori relativi alla rete ed essere facile da configurare, in quest'epoca, in cui le applicazioni sono distribuite sulla rete e si ridimensionano rapidamente, gestire gruppi di file, memorizzati localmente su macchine diverse, può non essere una scelta pratica. Al contrario, questo approccio può essere visto come un modo pratico per creare una copia di backup su file dei log, già aggregati su una destinazione di rete.

## Un esempio di basicConfig()

Di seguito viene presentato un esempio che utilizza basicConfig() con alcuni parametri di default per loggare su file. Il livello configurati è DEBUG; ciò permette di scrivere log con livello di severità pari o maggiore a DEBUG. Come informazione extra, sono provviste alcune regole di formattazione.

```
import logging

logging.basicConfig(level=logging.DEBUG,
                    filename='basic_config_test1.log',
                    format='%(asctime)s %(levelname)s:%(message)s')


def basic_config_test1():
   logging.debug("debug log test")
   logging.error("error log test")
```

Lanciando il codice, un nuovo basic_config_test1.log file viene generato con il seguente contenuto, fatta eccezione per le date, che dipendono chiaramente dal tempo di esecuzione.

```
2020-05-08 17:19:29,220 DEBUG:debug log test
2020-05-08 17:19:29,221 ERROR:error log test
```

NOTA: il metodo basicConfig() si riferisce alla configurazione del root logger e sortisce il suo effetto soltanto la prima volta in cui viene chiamato a runtime.

DEBUG-level log sono ora visibili, e tali log sono stampati con la seguente struttura personalizzata:

%(asctime)s: ora e giorno in formato locale dell'esecuzione della riga
%(levelname)s: livello di severità del messaggio
%(message)s: messaggio vero e proprio

Per completare il tutto ecco una tabella descrittiva delle opzioni del metodo basicConfig():

| Format | Description | Example |
|---|---|---|
| *filename* | Specifies that a FileHandler should be created, using the specified filename, rather than a StreamHandler | /var/logs/logs.txt |
| *format* | Use the specified format string for the handler | "'%(asctime)s %(message)s'" |
| *datefmt* | Use the specified datetime format | "%H:%M:%S" |
| *level* | Set the root logger level to the specified level | "INFO" |

Questo è un modo semplice e pratico per configurare piccoli script. Seguendo le best practice di Python raccomandiamo di gestire **un'istanza del logger per ogni modulo dell'applicazione**, ma è comprensibile che l'utilizzo delle sole funzionalità del metodo basicConfig() possa risultare impegnativo e poco pulito. Ora ci concentreremo sulle tecniche per migliorare la soluzione di logging di base.

## Best practice: una istanza di logging per ogni modulo

Seguendo gli standard sulle best practice, possiamo impostare una soluzione a singolo logger per istanza. Un buon approccio, che si adatta molto bene, è facile da configurare e dunque dovrebbe essere preso in considerazione quando si ha a che fare con **applicazioni di grandi dimensioni** che devono essere ridimensionate, è sfruttare il metodo getLogger() integrato.

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

Il metodo presentato crea un nuovo logger, differente da quello di root. L'argomento __name__ corrisponde al fully qualified name del modulo dal quale il metodo viene richiamato. Questo permette inoltre di tenere in considerazione il nome del logger come parte integrante di una riga di

log, creando dinamicamente il nome del logger sulla base del modulo con il quale si sta lavorando in quel momento.

È possibile recuperare il nome del logger con %(name)s come mostrato nel seguente esempio.

```
# logging_test1.py

import logging

logging.basicConfig(level=logging.DEBUG,
                    filename='basic_config_test1.log',
                    format='%(asctime)s %(name)s %(levelname)s:%(message)s')

logger = logging.getLogger(__name__)


def basic_config_test1():
   logger.debug("debug log test 1")
   logger.error("error log test 1")


# logging_test2.py

import logging
import logging_test1

logger = logging.getLogger(__name__)



def basic_config_test2():
   logger.debug("debug log test 2")
   logger.error("error log test 2")
```

Ora si è creata una **configurazione decisamente migliore;** ogni modulo descrive se stesso all'interno del log stream e tutto risulta molto più chiaro. Ciononostante, il logger creato nel modulo logging_test2, utilizzerà la stessa configurazione del logger creato nel modulo logging_test1, ovvero la configurazione del root logger. Come già accennato, la seconda invocazione di basicConfig() non sortirà effetto. Di conseguenza, eseguire sia basic_config_test1() che basic_config_test2() risulterà nella creazione di un singolo file basic_config_test1.log con il seguente contenuto.

```
2020-05-09 19:37:59,607 logging_test1 DEBUG:debug log test 1
2020-05-09 19:37:59,607 logging_test1 ERROR:error log test 1
2020-05-09 19:38:05,183 logging_test2 DEBUG:debug log test 2
2020-05-09 19:38:05,183 logging_test2 ERROR:error log test 2
```

A seconda del contesto dell'applicazione, usare una singola configurazione per ogni logger istanziato nei moduli potrebbe non essere sufficiente. Nella prossima sezione verrà analizzato come generare e distribuire le configurazioni di logging tra moduli diversi mediante fileConfig() o dictConfig().

## Usare fileConfig() e dictConfig()

Anche se utilizzare basicConfig() rappresenta un modo rapido per iniziare a organizzare il logging, le configurazioni basate su file o dizionario offrono più opzioni per andare incontro alle proprie esigenze, consentire più di un logger nella propria applicazione e inviare il log a destinazioni diverse in base a fattori specifici .

Questo è anche il metodo preferito di framework come Django e Flask per impostare il logging nei progetti. Nelle prossime sezioni, daremo un'occhiata più da vicino su come impostare una corretta configurazione basata su file o dizionario.

## fileConfig()

I file di configurazione devono aderire a questa struttura, contenente alcuni elementi chiave.

[loggers]: nome dei logger che si devono configurare.
[handlers]: handler da usare per logger specifici (es. consoleHandler, fileHandler).
[formatters]: il formato da applicare ad ogni logger.

Ogni sezione del file (denominata al plurale) dovrebbe includere una o più chiavi, separate da virgola, da collegare alla configurazione principale:

```
[loggers]
keys=root,secondary
```

The keys are used to traverse the file and read appropriate configurations for each section. The keys are defined as [_], where the section name is logger, handler, or formatter which are predefined as said before.

Le chiavi sono utilizzate per attraversare il file e leggere le configurazioni appropriate per ogni singola sezione. Le chiavi sono definite secondo la forma [_], dove può valere logger, handler o formatter come descritto poc'anzi.

Un semplice file di configurazione per il logging (logging.ini) è mostrato di seguito.

```
[loggers]
keys=root,custom

[handlers]
keys=fileHandler,streamHandler

[formatters]
keys=formatter

[logger_root]
level=DEBUG
handlers=fileHandler

[logger_custom]
level=ERROR
```

```
handlers=streamHandler

[handler_fileHandler]
class=FileHandler
level=DEBUG
formatter=formatter
args=("/path/to/log/test.log",)

[handler_streamHandler]
class=StreamHandler
level=ERROR
formatter=formatter

[formatter_formatter]
format=%(asctime)s %(name)s - %(levelname)s:%(message)s
```

Le best practice di Python raccomandano di mantenere **un solo handler per logger,** basandosi sul concetto di eredità del logger per la propagazione delle proprietà. Ciò significa che è sufficiente specificare una configurazione in un logger padre per averla ereditata da tutti i logger figli senza doverla impostare in ognuno di essi. Un buon esempio di ciò è un uso intelligente del logger root come genitore.

Ritornando al discorso principale, una volta preparato un file di configurazione, è possibile collegarlo ad un logger con queste semplici righe di codice:

```
import logging.config

logging.config.fileConfig('/path/to/logging.ini',
                          disable_existing_loggers=False)
logger = logging.getLogger(__name__)
```

Questo esempio include il parametro disable_existing_loggers valorizzato a False, garantendo che tutti i logger preesistenti non vengano rimossi quando lo snippet di codice viene eseguito; questo è indubbiamente un buon consiglio, perchè molti moduli sono soliti definire un root logger ancora prima che fileConfig() venga chiamato.

## dictConfig()

L'esempio di configurazione del logger così come visto finora si può descrivere anche mediante una comune struttura a dizionario. Tale dizionario seguirà la struttura presentata già con fileConfig() con sezioni concernenti logger, handler e formatter più la presenza di alcuni parametri globali di utilità. Di seguito un esempio:

```
import logging.config


config = {
   "version": 1,
   "disable_existing_loggers": False,
   "formatters": {
       "standard": {
```

```
            "format": "%(asctime)s %(name)s %(levelname)s %(message)s",
            "datefmt": "%Y-%m-%dT%H:%M:%S%z",
        }
    },
    "handlers": {
        "standard": {
            "class": "logging.StreamHandler",
            "formatter": "standard"
        }
    },
    "loggers": {
        "": {
            "handlers": ["standard"],
            "level": logging.INFO
        }
    }
}
```

Per applicare questa configurazione è sufficiente passare questo oggetto come parametro del dict config:

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

dictConfig() di base disabilita gli altri logger, a meno che disable_existing_loggers sia valorizzato a False così come fileConfig()nell'esempio precedente.

Tale configurazione può anche essere salvata in un file YAML e da lì configurata. Molti sviluppatori spesso preferiscono utilizzare dictConfig() rispetto a fileConfig(), questo perchè ha molti più spunti di configurazione, più facile da leggere e più mantenibile.

## Formattare i log secondo lo standard JSON

Grazie al modo in cui è progettato, è facile estendere il modulo di logging. Poiché l'obiettivo è automatizzare il logging e integrarlo con Kibana, si desidera un formato più adatto per l'aggiunta di proprietà personalizzate e più compatibile con i carichi di lavoro personalizzati. Quindi è necessario configurare un **formattatore JSON.**

Se si vuole, è possibile loggare in formato JSON creando un formattatore personalizzato che trasforma i record di log in una stringa codificata JSON:

```
import logging
import json


class JsonFormatter:
    def format(self, record):
        formatted_record = dict()

        for key in ['created', 'levelname', 'pathname', 'msg']:
            formatted_record[key] = getattr(record, key)

        return json.dumps(formatted_record, indent=4)
```

```
handler = logging.StreamHandler()
handler.formatter = JsonFormatter()

logger = logging.getLogger(__name__)
logger.addHandler(handler)

logger.error("Test")
```

Se non si vuole creare da zero il proprio formattatore custom, si può fare riferimento a diverse librerie, sviluppate dalla community Python, che permettono di convertire i propri log in formato JSON. Una di queste è python-json-logger . Per prima cosa bisogna installare la libreria nel proprio ambiente di sviluppo:

```
pip install python-json-logger
```

Ora è possibile aggiornare la configurazione del dizionario per modificare un formattatore esistente o creare un nuovo formattatore da zero che generi stringhe in JSON come nell'esempio seguente. Per usare un formattatore JSON basta specificare pythonjsonlogger.jsonlogger.JsonFormatter come valore della proprietà "class". Nella chiave "format" si possono definire gli attributi da aggiungere ai record di log:

```
import logging.config


config = {
   "version": 1,
   "disable_existing_loggers": False,
   "formatters": {
       "standard": {
           "format": "%(asctime)s %(name)s %(levelname)s %(message)s",
           "datefmt": "%Y-%m-%dT%H:%M:%S%z",
       },
 "json": {
    "format": "%(asctime)s %(name)s %(levelname)s %(message)s",
    "datefmt": "%Y-%m-%dT%H:%M:%S%z",
    "class": "pythonjsonlogger.jsonlogger.JsonFormatter"
 }
   },
   "handlers": {
       "standard": {
           "class": "logging.StreamHandler",
           "formatter": "json"
       }
   },
   "loggers": {
       "": {
           "handlers": ["standard"],
           "level": LogLevel.INFO.value
       }
   }
}
```

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

I log inviati alla console, attraverso l'handler standard, verranno scritti in formato JSON.

Una volta inclusa la classe pythonjsonlogger.jsonlogger.JsonFormatter nel proprio file di configurazione di logging, la funzione dictConfig() è in grado di creare un'istanza di questo formattatore, ammesso che il codice venga eseguito in un ambiente di lavoro nel quale è possibile importare il modulo python-json-logger.

## Aggiungere informazioni contestuali ai propri log

Se si ha la necessità di aggiungere un pò di contesto ai propri log, il modulo di logging di Python fornisce la possibilità di **aggiungere attributi custom.** Un modo elegante per arricchire i propri log richiede l'uso della classe LoggerAdapter.

```
logger = logging.LoggerAdapter(logger, {"custom_key1": "custom_value1", "custom_key2"
: "custom_value2"})
```

Ciò aggiunge effettivamente attributi personalizzati a tutti i record che attraversano quel logger. Si noti che ciò influisce su tutti i record di log nell'applicazione, comprese le librerie o altri framework che potrebbero essere in uso e per i quali si stanno emettendo log. Questo snippet può essere usato per aggiungere attributi quali ID di richiesta univoci su tutte le righe di log, per tenere traccia delle richieste, o per aggiungere ulteriori informazioni contestuali.

## Python: gestione delle exception e tracebacks

**Di default** gli errori non includono nessuna informazione di traceback. Il log mostrerà semplicemente l'eccezione come errore, senza dettagli aggiuntivi. Per essere sicuri che logging.error() mostri l'intero traceback, è necessario aggiungere la proprietà exc_info valorizzata a True. Per mostrare la differenza, si provi a loggare una eccezione con e senza exc_info.

```
import logging.config


config = {
  "version": 1,
  "disable_existing_loggers": False,
  "formatters": {
      "standard": {
          "format": "%(asctime)s %(name)s %(levelname)s %(message)s",
          "datefmt": "%Y-%m-%dT%H:%M:%S%z",
      },
      "json": {
          "format": "%(asctime)s %(name)s %(levelname)s %(message)s",
          "datefmt": "%Y-%m-%dT%H:%M:%S%z",
          "class": "pythonjsonlogger.jsonlogger.JsonFormatter"
      }
  },
  "handlers": {
```

```
        "standard": {
            "class": "logging.StreamHandler",
            "formatter": "json"
        }
    },
    "loggers": {
        "": {
            "handlers": ["standard"],
            "level": logging.INFO
        }
    }
}

logging.config.dictConfig(config)
logger = logging.getLogger(__name__)


def test():
    try:
        raise NameError("fake NameError")
    except NameError as e:
        logger.error(e)
        logger.error(e, exc_info=True)
```

Se eseguiamo il metodo test(), quest'ultimo genererà il seguente output:

```
{"asctime": "2020-05-10T16:43:12+0200", "name": "logging_test", "levelname": "ERROR",
"message": "fake NameError"}

{"asctime": "2020-05-10T16:43:12+0200", "name": "logging_test", "levelname": "ERROR",
"message": "fake NameError", "exc_info": "Traceback (most recent call last):\n  File
 \"/Users/aleric/Projects/logging-test/src/logging_test.py\", line 39, in test\n    ra
ise NameError(\"fake NameError\")\nNameError: fake NameError"}
```

Come possiamo vedere, la prima riga non fornisce molte informazioni sull'errore a parte un messaggio generico fake NameError; invece, la seconda riga, aggiungendo la proprietà exc_info = True, mostra il traceback completo, che include informazioni sui metodi coinvolti e numeri di riga da seguire per vedere dove è stata sollevata l'eccezione.

In alternativa si può ottenere lo stesso risultato usando logger.exception() da un exception handler come nell'esempio seguente:

```
def test():
    try:
        raise NameError("fake NameError")
    except NameError as e:
        logger.error(e)
        logger.exception(e)
```

Questo snippet permette di recuperare le stesse informazioni di traceback dell'esempio precedente. Ciò ha anche il vantaggio di impostare il livello di priorità del logging su logging.ERROR.

Indipendentemente dal metodo utilizzato per acquisire il traceback, la disponibilità di tutte le informazioni sulle eccezioni complete nei log è fondamentale per il monitoraggio e la risoluzione dei problemi delle prestazioni delle applicazioni.

## Catturare eccezioni non gestite

Anche pensando a diversi scenari, è chiaro che non è possibile gestire ogni possibile eccezione che la propria applicazione può generare. È comunque possibile loggare eccezioni non gestite in modo da poterle esaminare in seguito.

Un'eccezione risulta non gestita se generata al di fuori di un blocco try / except o se è di un tipo diverso da quelli specificati in uno o più statement except.

Se un'eccezione non viene gestita nel metodo che in cui viene generata, viene trasmessa al metodo superiore che può, eventualmente, contenere la logica di gestione della stessa. Questo meccanismo trasmette l'eccezione fino al metodo main, ammesso che i metodi sottostanti non contengano la logica di gestione di quella tipologia di eccezione.

Se non viene gestita l'interprete invocherà sys.excepthook(). Le informazioni di traceback non verranno loggate se non esplicitamente specificato all'interno di un handler non predefinito (ad esempio un fileHandler).

È possibile utilizzare **la libreria traceback di Python** per formattare il traceback ed includerlo nel log record.

Per esaminare il comportamento descritto sopra, modifichiamo la funzione di esempio in modo che venga generata un'eccezione non gestita dal blocco try / except:

```
import logging.config
import traceback

config = {
  "version": 1,
  "disable_existing_loggers": False,
  "formatters": {
      "standard": {
          "format": "%(asctime)s %(name)s %(levelname)s %(message)s",
          "datefmt": "%Y-%m-%dT%H:%M:%S%z",
      },
      "json": {
          "format": "%(asctime)s %(name)s %(levelname)s %(message)s",
          "datefmt": "%Y-%m-%dT%H:%M:%S%z",
          "class": "pythonjsonlogger.jsonlogger.JsonFormatter"
      }
  },
  "handlers": {
      "standard": {
          "class": "logging.StreamHandler",
          "formatter": "json"
      }
  },
  "loggers": {
```

```
        "": {
            "handlers": ["standard"],
            "level": logging.INFO
        }
    }
}

logging.config.dictConfig(config)
logger = logging.getLogger(__name__)


def test():
    try:
        raise OSError("fake OSError")
    except NameError as e:
        logger.error(e, exc_info=True)
    except:
        logger.error("Uncaught exception: %s", traceback.format_exc())
```

Se eseguiamo il codice verrà lanciata un'eccezione di tipo OSError non gestita dalla logica di try-except. Nonostante ciò, grazie alla porzione di codice definita nel secondo blocco except, l'eccezione verrà comunque loggata:

```
{"asctime": "2020-05-10T17:04:05+0200", "name": "logging_test", "levelname": "ERROR",
"message": "Uncaught exception: Traceback (most recent call last):\n  File \"/Users/al
eric/Projects/logging-test/src/logging_test4.py\", line 38, in test\n    raise OSError
(\"fake OSError\")\nOSError: fake OSError\n"}
```

Loggare il traceback completo di ogni eccezione, gestita o meno, garantisce una **visibilità cruciale** sugli errori generati in real-time, e permette di investigare le cause generanti.

Questa è la fine della parte uno di due del nostro articolo. Nella prossima parte vedremo **come aggregare i log usando Amazon Kinesis Data Firehose ed Amazon ElasticSearch Service, e Kibana per analizzare i log in una dashboard** centralizzata ed altamente customizzabile.
Rimanete sintonizzati 🙂


## beSharp

Dal 2011 beSharp guida le aziende italiane sul Cloud. Dalla piccola impresa alla grande multinazionale, dal manifatturiero al terziario avanzato, aiutiamo le realtà più all'avanguardia a realizzare progetti innovativi in campo IT.

### Get in touch

beSharp.it
proud2becloud@besharp.it