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

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

beSharp | 29 Maggio 2020

In questa seconda parte del nostro viaggio alla scoperta dei segreti e delle best practice del logging in Python (se vi siete persi la prima parte, eccola qui!) si farà un passo ulteriore: gestire più istanze applicative (e quindi più stream di log), cosa piuttosto comune in scenari riguardanti progetti cloud, al fine di aggregare i log usando Amazon Kinesis Data Firehose, Amazon ElasticSearch Service e Kibana. Incominciamo!

E' il momento di aggregare i log!

Nei casi in cui si voglia monitorare un'applicazione complessa, sforzarsi di raccogliere log distribuiti per comprendere cosa è andato storto con il codice non è un'idea particolarmente praticabile.

Supponiamo che si abbia implementato un'API REST serverless tramite AWS API Gateway integrata via proxy con AWS Lambda Functions, scritta in Python, per ciascuno degli endpoint che si ha definito. Dato ciò che abbiamo già trattato in precedenza, in questo caso i log verranno presumibilmente scritti nei flussi di AWS CloudWatch tramite StreamHandler.

E se invece di cercare i record di log all'interno dei flussi di log di CloudWatch, si desiderasse analizzarli da una dashboard centralizzata? Bene, in questo caso la risposta si chiama stack EKK (Amazon **E**lasticsearch Service, Amazon **K**inesis Data Firehose e **K**ibana).

Prima di entrare nei dettagli della configurazione dello stack, introduciamo il ruolo di ciascuno degli attori dello stack. Quella che segue è solo un'introduzione ai servizi utilizzati per questa soluzione di log, ricerca e analisi. Se si desiderasse ottenere maggiori informazioni su ciascuno dei servizi utilizzati, invitiamo a consultare la documentazione a loro dedicata.

Attori dello stack EKK

Amazon Elasticsearch Service è un servizio gestito che consente di distribuire, gestire e ridimensionare un cluster Elasticsearch nel proprio account AWS. Fornisce un motore di ricerca e analisi che sfruttabile per monitorare i log della propria applicazione in tempo reale.

Amazon Elasticsearch Service ha un'integrazione nativa con Kibana. Kibana è uno strumento che offre una dashboard facile da usare in cui è possibile monitorare ed eseguire il debug dell'applicazione in modo centralizzato.

Amazon Kinesis Data Firehose è il servizio che funge da ponte tra i generatori di log e il cluster Elasticsearch. Kinesis Data Firehose consente di caricare i dati di streaming su uno o più target specifici. Nella soluzione proposta in questo articolo, **Kinesis Data Firehose** viene utilizzato per lo streaming dei record di log prodotti da componenti dell'applicazione diversi e distribuiti su un cluster Elasticsearch e su un bucket S3, entrambi ospitati su un account AWS. Il bucket S3 viene utilizzato come backup dei record di registro e può essere utilizzato per recuperare dati storici.

Per quanto riguarda l'ecosistema Python, si può fare affidamento sull'**SDK di AWS boto3** per trasmettere in streaming i registri locali direttamente a un delivery stream di Kinesis Data Firehose. Combinando il modulo di registrazione di Python con l'SDK di boto3, si può eseguire lo streaming dei propri log su Kinesis Data Firehose. Nella sezione successiva vedremo come implementare il gestore di un modulo di log Python che caricherà una versione JSON dei record di log in un delivery stream di Kinesis Data Firehose.

Estendere il modulo di Logging di Python

Grazie alla natura estensibile del modulo di logging di Python, è possibile implementare un handler personalizzato che soddisfi le nostre esigenze. In questa sezione illustreremo come implementare uno StreamHandler che trasmetta i dati di log a un delivery stream di Kinesis Data Firehose.

Ecco l'implementazione:

```
import boto3
import logging


class KinesisFirehoseDeliveryStreamHandler(logging.StreamHandler):

   def __init__(self):
       # By default, logging.StreamHandler uses sys.stderr if stream parameter is not
 specified
       logging.StreamHandler.__init__(self)

       self.__firehose = None
       self.__stream_buffer = []

       try:
           self.__firehose = boto3.client('firehose')
       except Exception:
           print('Firehose client initialization failed.')
```

```
        self.__delivery_stream_name = "logging-test"

    def emit(self, record):
        try:
            msg = self.format(record)

            if self.__firehose:
                self.__stream_buffer.append({
                    'Data': msg.encode(encoding="UTF-8", errors="strict")
                })
            else:
                stream = self.stream
                stream.write(msg)
                stream.write(self.terminator)

            self.flush()
        except Exception:
            self.handleError(record)

    def flush(self):
        self.acquire()

        try:
            if self.__firehose and self.__stream_buffer:
                self.__firehose.put_record_batch(
                    DeliveryStreamName=self.__delivery_stream_name,
                    Records=self.__stream_buffer
                )

                self.__stream_buffer.clear()
        except Exception as e:
            print("An error occurred during flush operation.")
            print(f"Exception: {e}")
            print(f"Stream buffer: {self.__stream_buffer}")
        finally:
            if self.stream and hasattr(self.stream, "flush"):
                self.stream.flush()

            self.release()
```

Come si può notare, andando sullo specifico, l'esempio mostra una classe, KinesisFirehoseDeliveryStreamHandler, che eredita il comportamento nativo della classe StreamHandler. I metodi di StreamHandler modificati per questo esempio sono emit e flush.
Il metodo emit è responsabile dell'invocazione del metodo di format, dell'aggiunta di record di log allo stream e del metodo di flush. La modalità di formattazione dei dati di log dipende dal tipo di formattatore configurato per il gestore. Indipendentemente dalla modalità di formattazione, i dati di log verranno aggiunti all'array __stream_buffer o, nel caso in cui qualcosa sia andato storto durante l'inizializzazione del client Firehose, al flusso predefinito, ovvero sys.stderr.

Il metodo flush è responsabile dello streaming dei dati direttamente nel delivery stream di Kinesis Data Firehose attraverso l'API put_record_batch. Una volta che i record vengono trasmessi in streaming sul Cloud, lo _stream_buffer locale verrà cancellato. L'ultimo passaggio del metodo flush consiste nel flushing dello stream di default.

Questa implementazione è puramente illustrativa ma cionondimeno solida per cui ci si senta liberi di copiare e personalizzare lo snippet in base alle proprie esigenze specifiche.

Dopo aver incluso KinesisFirehoseDeliveryStreamHandler nella propria codebase, si deve poi aggiungerlo alla configurazione dei logger. Vediamo come cambia la configurazione del dizionario precedente nell'introdurre il nuovo gestore.

```
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"
      },
      "kinesis": {
          "class": "KinesisFirehoseDeliveryStreamHandler.KinesisFirehoseDeliveryStream
Handler",
          "formatter": "json"
      }
  },
  "loggers": {
      "": {
          "handlers": ["standard", "kinesis"],
          "level": logging.INFO
      }
  }
}
```

Per includere il nuovo handler personalizzato nella propria configurazione, è sufficiente aggiungere una voce "kinesis" al dizionario "handlers" e una voce "kinesis" nell'array "handlers" del logger di root.

Nella voce "kinesis" del dizionario "handlers" dovremmo specificare la classe del handler personalizzato e il formatter utilizzato da quest'ultimo per formattare i record di log.

Aggiungendo una voce "kinesis" all'array "handlers" del logger di root, si sta indicando a quest'ultimo di scrivere record di log sia in console che nel delivery stream di Kinesis Data Firehose.

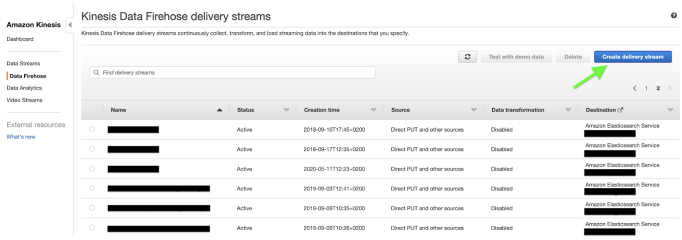
PS: il logger di root è identificato da "" nella sezione "loggers".

Questo è tutto ciò che serve per la configurazione del producer dei dati di log di Kinesis Data Firehose. Concentriamoci ora sull'infrastruttura dietro l'API put_record_batch, quella utilizzata da KinesisFirehoseDeliveryStreamHandler per lo streaming dei record di log sul cloud.
Dietro le quinte dell'API put_record_batch

I componenti dell'architettura necessari per aggregare i record di log dell'applicazione e renderli disponibili e ricercabili da una dashboard centralizzata sono i seguenti:

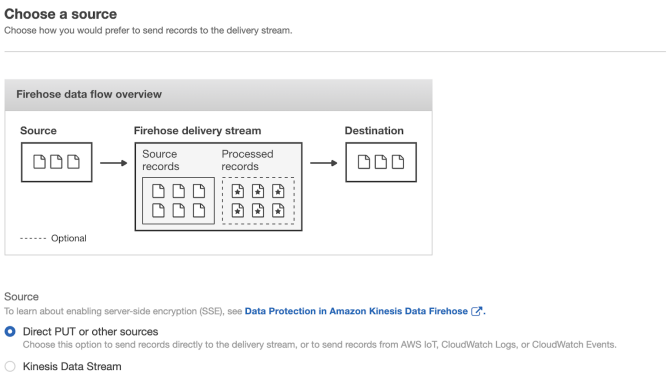
un delivery stream di Kinesis Data Firehose;
un cluster del servizio Amazon Elasticsearch.

Per creare un delivery stream di Kinesis Data Firehose, passiamo alla dashboard Kinesis della console di gestione AWS. Dal menù a sinistra, selezioniamo Data Firehose. Una volta selezionato, dovremmo visualizzare un elenco di data stream presenti in una regione specifica del proprio account AWS. Per impostare un nuovo delivery stream, faremo clic sul pulsante Create delivery stream nell'angolo in alto a destra della pagina.



Nella procedura guidata di Create delivery stream ci verrà chiesto di configurare l'origine del delivery stream, il processo di trasformazione, la destinazione e altre impostazioni come le autorizzazioni necessarie a Kinesis Data Firehose per caricare i dati di streaming nelle destinazioni specificate.

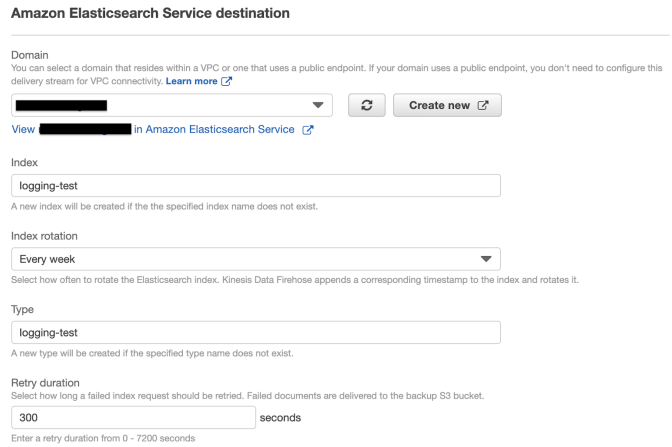
Poiché stiamo caricando i dati direttamente dal nostro logger tramite l'SDK di boto3, dobbiamo scegliere Direct PUT or other sources come sorgente del delivery stream.



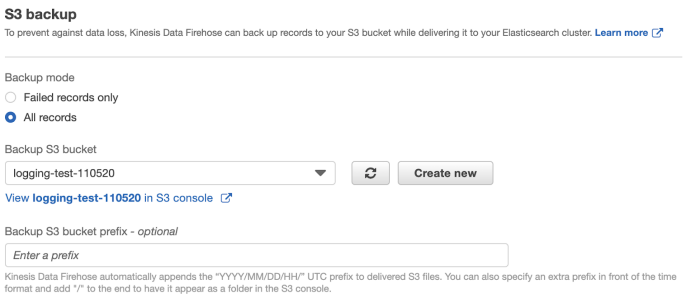
Lasciamo le opzioni "transform" e "convert" disabilitate in quanto non fondamentali ai risultati presentati in questo articolo.

Il terzo step del wizard richiede di specificare le destinazioni del delivery stream. Assumendo che si sia già creato un cluster di Amazon Elasticsearch nel proprio account AWS, questo verrà specificato come destinazione primaria, indicando l'index name di Elasticsearch, la rotation frequency, il

mapping type e la retry duration, ovvero per quanto a lungo una richiesta fallita deve essere ritentata.



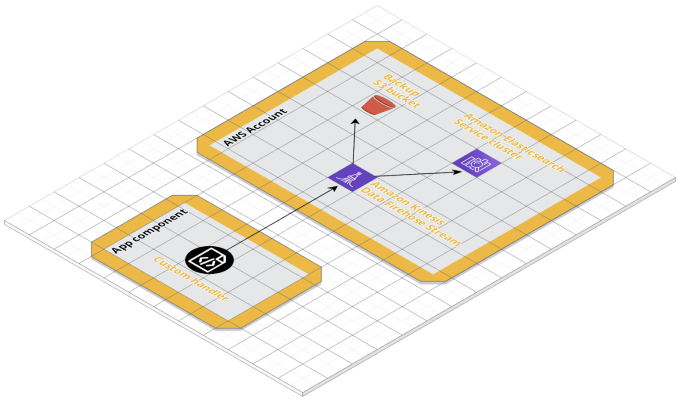
Come destinazione secondaria del nostro delivery stream, imposteremo un bucket S3. Come già accennato in precedenza, questo bucket conterrà registri storici non soggetti alla logica di rotazione dell'indice di Elasticsearch.



Lasceremo disabilitate la compressione S3, la crittografia S3 e la registrazione degli errori e ci concentreremo sulle autorizzazioni. Quest'ultima sezione richiede di specificare o creare un nuovo ruolo IAM con una politica che consenta a Kinesis Data Firehose di trasmettere i dati alle destinazioni specificate. Facendo clic su Create new verremo guidati nella creazione di un ruolo IAM con il set di criteri di autorizzazione richiesto.

Log record streaming test

Una volta che il delivery stream è creato, possiamo testare finalmente se il codice e l'architettura sono stati correttamente integrati. Il seguente schema illustra gli attori in gioco:



Dalla propria macchina locale si andrà a simulare una applicazione che andrà a caricare i dati direttamente su un Kinesis Data Firehose delivery stream. Per questo test si andrà ad utilizzare la configurazione a dizionario che già include il KinesisFirehoseDeliveryStreamHandler.

```
import logging.config

config = {...}

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


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

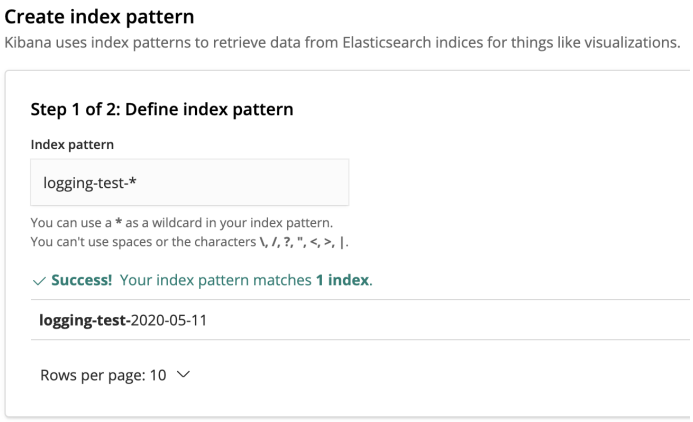
Eseguendo il test, un nuovo record di log verrà generato e scritto sia in console che sul delivery stream.
Di seguito l'output della console in fase di test:

```
{"asctime": "2020-05-11T14:44:44+0200", "name": "logging_test5", "levelname": "ERROR"
, "message": "fake NameError", "exc_info": "Traceback (most recent call last):\n  File
\"/Users/ericvilla/Projects/logging-test/src/logging_test5.py\", line 42, in test\n
raise NameError(\"fake NameError\")\nNameError: fake NameError"}
```

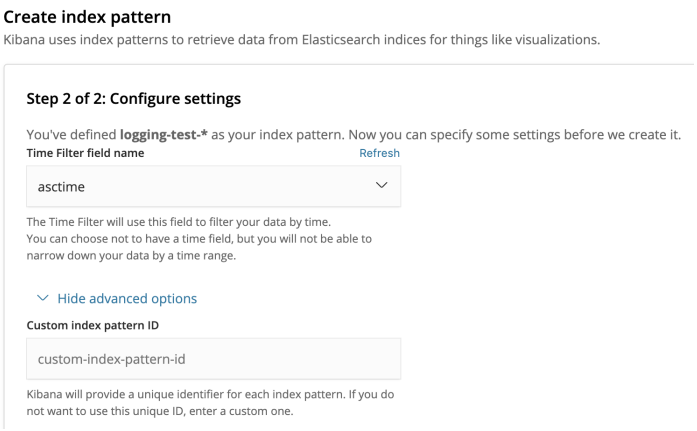
Beh, niente di nuovo. Ciò che ci si aspetterebbe oltre all'output della console è trovare anche il record di log nella nostra console di Kibana.

Per consentire la ricerca e l'analisi dei record di log direttamente da Kibana, è necessario creare un index pattern, utilizzato da Kibana per recuperare dati da specifici indici di Elasticsearch.

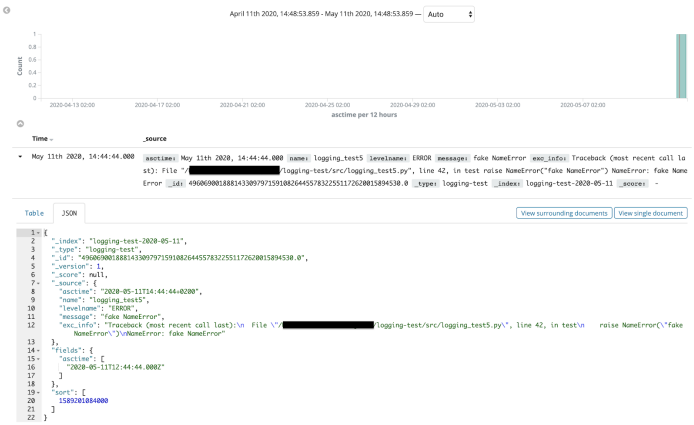
Il nome che abbiamo dato all'indice Elasticsearch è logging-test. Pertanto, gli indici verranno archiviati come logging-test-. Fondamentalmente, per far sì che Kibana recuperi i record di log da ciascun indice che inizi con logging-test-, si dovrà definire il pattern di log logging-test- *. Se il nostro KinesisFirehoseDeliveryStreamHandler avrà funzionato come previsto, l'index pattern dovrebbe corrispondere ad un nuovo indice.



Per filtrare i record di log per orario, possiamo usare la chiave asctime che il formatter JSON avrà aggiunto a tale record.



Una volta che L'index pattern è creato, possiamo finalmente ricercare ed analizzare i log direttamente dalla console di Kibana!



È possibile personalizzare ulteriormente l'esperienza di ricerca e analisi dei record di log, per eseguire il debug dell'applicazione in modo più efficiente, aggiungendo filtri e creando dashboard.

Detto tutto questo, qui si conclude il nostro viaggio alla scoperta del modulo di logging di Python, delle best practices e delle tecniche per aggregare log distribuiti. Speriamo vivamente che vi sia piaciuto leggere questo articolo e che abbiate potuto impare qualche nuovo trucco. Fino al prossimo articolo, state al sicuro 🙂

Leggi la Parte 1

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