Utilizzo di metadati per report personalizzati con API e Python

Sommario

Introduzione

In questo documento viene descritto come utilizzare i metadati insieme alle API per creare report personalizzati all'interno di uno script Python.

Prerequisiti

Requisiti

Cisco raccomanda la conoscenza dei seguenti argomenti:

- CloudCenter
- Python

Componenti usati

Il documento può essere consultato per tutte le versioni software o hardware.

Le informazioni discusse in questo documento fanno riferimento a dispositivi usati in uno specifico ambiente di emulazione. Su tutti i dispositivi menzionati nel documento la configurazione è stata ripristinata ai valori predefiniti. Se la rete è operativa, valutare attentamente eventuali conseguenze derivanti dall'uso dei comandi.

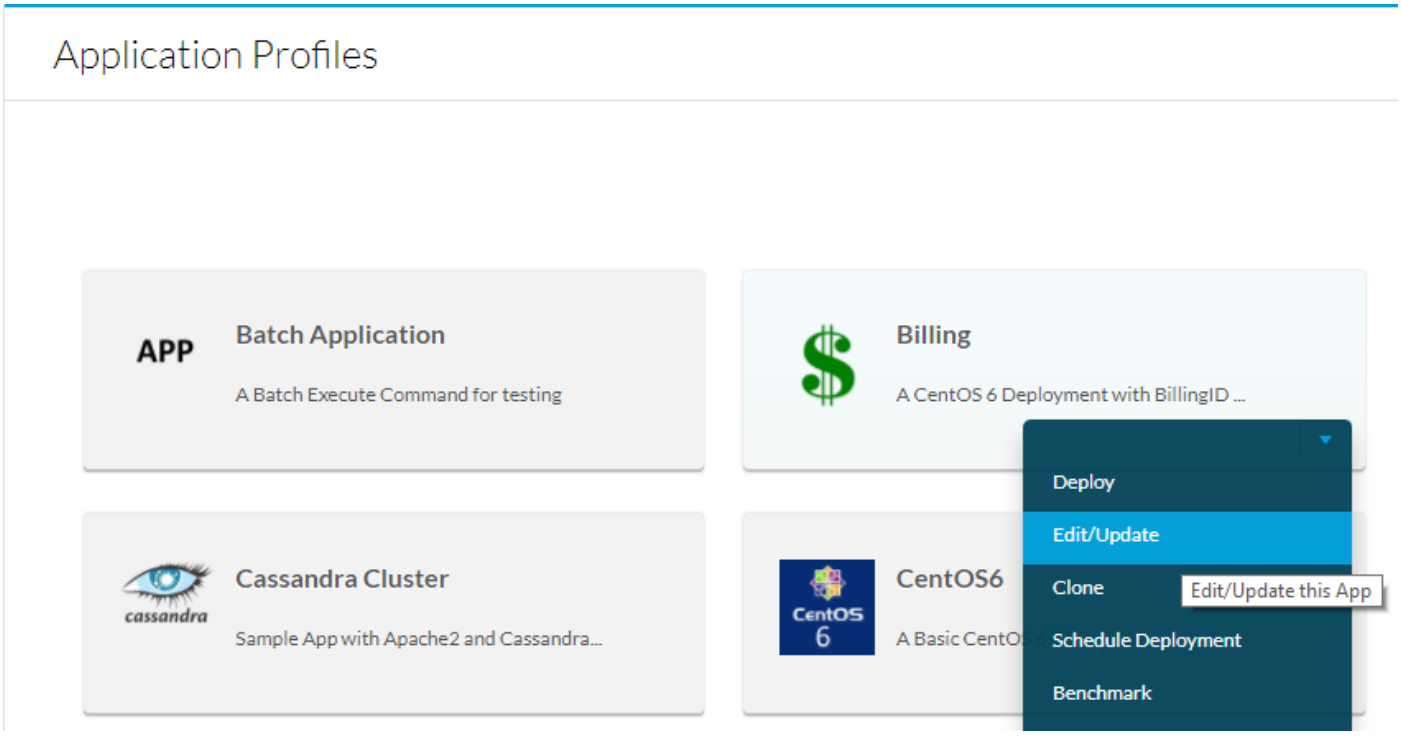
Premesse

CloudCenter fornisce alcuni report preconfigurati, ma non consente di creare report basati su filtri personalizzati. Per utilizzare le API per recuperare le informazioni direttamente dal database, insieme ai metadati associati ai processi, è possibile consentire l'utilizzo di report personalizzati.

Impostare i metadati

I metadati devono essere aggiunti a livello di applicazione, quindi ogni applicazione che deve essere rilevata con l'utilizzo del report personalizzato dovrà essere modificata.

A tale scopo, passare a **Profili applicazione**, quindi selezionare l'elenco a discesa dell'app da modificare e selezionare **Modifica/Aggiorna** come mostrato nell'immagine.



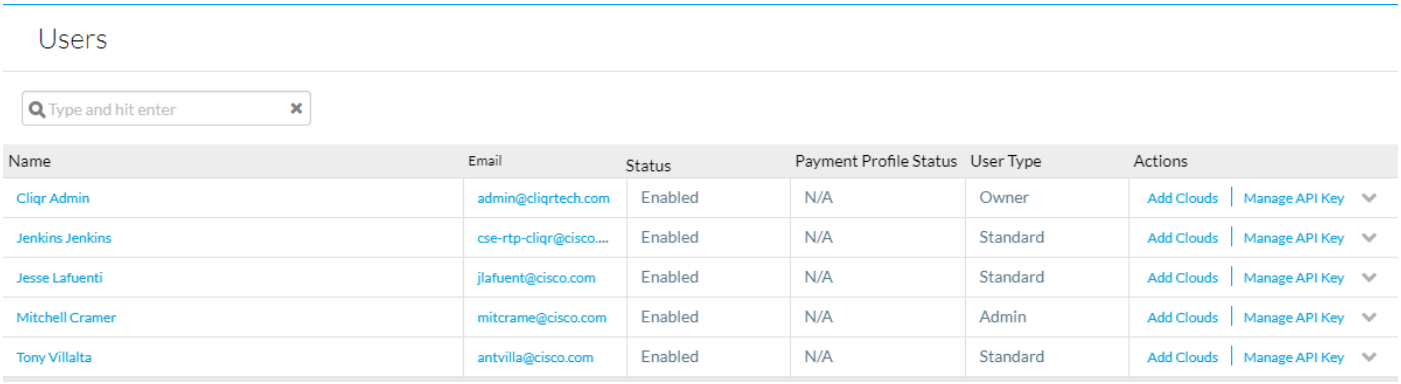
Scorrere fino alla fine di **Informazioni di base** e aggiungere un tag Metadata, ad esempio **BillingID**, se questi metadati devono essere compilati dall'utente, renderli obbligatori e modificabili. Se si tratta solo di una macro, immettere il valore predefinito e non modificarlo. Dopo aver compilato i metadati, selezionare **Add**, quindi **Save App** (Salva applicazione) come mostrato nell'immagine.

Raccogli chiavi API

Per elaborare le chiamate API, saranno necessari il nome utente e le chiavi API. Queste chiavi forniscono lo stesso livello di accesso dell'utente, pertanto se si desidera aggiungere nel report tutte le distribuzioni degli utenti, è consigliabile utilizzarle per ottenere l'amministratore delle chiavi API dei tenant. Se è necessario registrare più subtenant contemporaneamente, il tenant radice deve accedere a tutti gli ambienti di distribuzione oppure saranno necessarie le chiavi API di tutti gli amministratori subtenant.

Per ottenere le chiavi API, passare a **Amministrazione > Utenti > Gestisci chiave API**, copiare il

nome utente e la chiave per gli utenti richiesti.



Creazione del report personalizzato

Prima di creare lo script python che crea il report, verificare che python e pip siano installati. Eseguire quindi **pip install tabulate**, tabulate è una libreria che gestisce automaticamente la formattazione del report.

A questa guida sono allegati due report di esempio, il primo si limita a raccogliere informazioni su tutte le distribuzioni e a inviarle in una tabella. Il secondo utilizza le stesse informazioni per creare un report personalizzato con l'utilizzo dei metadati BillingID. Questo script viene spiegato in dettaglio da utilizzare come guida.

```
import datetime
import json
import sys
import requests
##pip install tabulate
from tabulate import tabulate
from operator import itemgetter
from decimal import Decimal
```

datetime viene utilizzato per calcolare con precisione la data, in modo da creare un report degli ultimi X giorni.

il formato json viene utilizzato per analizzare i dati json, l'output delle chiamate api.

sys viene utilizzato per le chiamate di sistema.

Le richieste vengono utilizzate per semplificare la creazione di richieste Web per le chiamate API.

tabulate viene utilizzato per formattare automaticamente la tabella.

itemgetter viene utilizzato come iteratore per ordinare una tabella 2D.

Decimale viene utilizzato per arrotondare il costo a due cifre decimali.

```
if(len(sys.argv)==1):
        days = -1
elif(len(sys.argv)==2):
        try:
```

```
                days = int(sys.argv[1])
                if(days < 1):
                        raise ValueError('Less than 1')
                start=datetime.datetime.now()+datetime.timedelta(days*-1)
        except ValueError:
                print("Number of days must be an integer greater than 0")
                exit()
else:
        print("Enter number of days to report on, or leave blank to report all time")
        exit()
```

Questa parte viene utilizzata per analizzare il parametro della riga di comando relativo al numero di giorni.

Se non sono presenti parametri della riga di comando (sys.argv ==1), il report verrà eseguito per tutto il tempo.

Se è presente un controllo dei parametri della riga di comando se è un numero intero maggiore o uguale a 1, in caso contrario viene restituito un errore.

Se sono presenti più parametri, viene restituito un errore.

```
departments = []
users = ['user1','user2','user3']
passwords = ['user1Key','user2Key','user3Key']
```

departments è l'elenco che conterrà l'output finale.

users è un elenco di tutti gli utenti che effettueranno le chiamate API. Se esistono più subtenant, ogni utente sarebbe l'amministratore di un subtenant diverso.

password è un elenco delle chiavi API degli utenti. L'ordine degli utenti e delle chiavi deve essere identico per poter utilizzare la chiave corretta.

```
for j in xrange(0,len(users)):
        jobs = []
        r = requests.get('https://ccm2.cisco.com/v1/jobs', auth=(users[j], passwords[j]),
headers={'Accept': 'application/json'})
        data = r.json()
        for i in xrange(0,len(data["jobs"])):
                test = datetime.datetime.strptime((data["jobs"][i]["startTime"]), '%Y-%m-%d
%H:%M:%S.%f')
                if(days != -1):
                        if(start < test):
                                jobs.append([data["jobs"][i]["id"],'None',
data["jobs"][i]["cost"]["totalCost"],data["jobs"][i]["status"],data["jobs"][i]["displayName"],da
ta["jobs"][i]["startTime"]])
                else:
                        jobs.append([data["jobs"][i]["id"],'None',
data["jobs"][i]["cost"]["totalCost"],data["jobs"][i]["status"],data["jobs"][i]["displayName"],da
ta["jobs"][i]["startTime"]])
        for id in jobs:
                q = requests.get('https://ccm2.cisco.com/v1/jobs/'+id[0], auth=(users[j],
passwords[j]), headers={'Accept': 'application/json'})
                data2 = q.json()
                id[2]=round(id[2],2)
                for i in xrange(0,len(data2["metadatas"])):
```

```
                if('BillingID' == data2["metadatas"][i]["name"]):
                        id[1]=data2["metadatas"][i]["value"]
        added=0
        for i in xrange(0,len(departments)):
                if(departments[i][0]==id[1]):
                        departments[i][1]+= 1
                        departments[i][2]+=id[2]
                        added=1
        if(added==0):
                departments.append([id[1],1,id[2]])
```

per j in xrange(0,len(users): ciclo for per l'iterazione in ogni utente definito nel blocco di codice precedente. si tratta del ciclo principale che gestisce tutte le chiamate API.

processi è un elenco temporaneo che verrà utilizzato per memorizzare le informazioni relative ai processi mentre vengono fascicolate nell'elenco.

r = richieste.get.... è la prima chiamata API, questa elenca tutti i processi. Per ulteriori informazioni, vedere Elenco processi.

I risultati vengono quindi memorizzati in formato json in **dati.**

for i in xrange(0,len(data["jobs"]): esegue un'iterazione in tutti i processi restituiti dalla chiamata API precedente.

L'ora di ogni processo viene estratta dal file json e convertita in un oggetto datetime, quindi viene confrontata con il parametro della riga di comando immesso per verificare se rientra nei limiti.

In caso affermativo, queste informazioni derivate dal file json vengono aggiunte all'elenco dei job: **id, totalCost, status, name, start time**. Non vengono utilizzate tutte queste informazioni e non vengono restituite tutte le informazioni. In Processi elenco vengono visualizzate tutte le informazioni restituite che possono essere aggiunte allo stesso modo.

Dopo aver eseguito l'iterazione di tutti i job restituiti da tale utente, si passa alla sezione **for id in jobs:** che scorre tutti i job eseguiti dopo il controllo della data di inizio.

q = request.get(..... è la seconda chiamata API, questa elenca tutte le informazioni correlate all'ID processo ottenuto dalla prima chiamata API. Per ulteriori informazioni, vedere Ottieni dettagli processo.

Il file json viene quindi archiviato in **data2.**

Il costo memorizzato in **id[2]** viene arrotondato al secondo decimale.

per i in xrange(0,len(data2["metadata"]): scorre tutti i metadati associati al processo.

Se sono presenti metadati denominati **BillingID**, vengono memorizzati nelle informazioni sul processo.

aggiunto è un flag utilizzato per determinare se il **BillingID** è già stato aggiunto all'elenco **dei reparti** o meno.

per i in xrange(0,len(departments): esegue iterazioni in tutti i reparti aggiunti.

Se l'OdL fa parte di un reparto già esistente, il conteggio dell'OdL viene iterato di uno e il costo viene aggiunto al costo totale per tale reparto.

In caso contrario, una nuova linea viene aggiunta ai reparti con un conteggio mansioni pari a 1 e un costo totale uguale al costo di un'unica mansione.

```
departments = sorted(departments, key=itemgetter(1))
print(tabulate(departments,headers=['Department','Number of Jobs','Total Cost']))
```

departments = ordinati(departments, key=itemgetter(1)) ordina i reparti in base al numero di processi.

print(tabulate(departments,headers=['Department','Number of Jobs', 'Total Cost']) stampa una tabella creata da una tabella con tre intestazioni.

Informazioni correlate

- API CloudCenter
- **Documentazione e supporto tecnico – Cisco Systems**