Verwenden von Metadaten für benutzerdefinierte Berichte mit APIs und Python

Inhalt

Einführung

In diesem Dokument wird beschrieben, wie Metadaten in Verbindung mit APIs verwendet werden, um einen benutzerdefinierten Bericht in einem Python-Skript zu erstellen.

Voraussetzungen

Anforderungen

Cisco empfiehlt, über Kenntnisse in folgenden Bereichen zu verfügen:

- CloudCenter
- Python

Verwendete Komponenten

Dieses Dokument ist nicht auf bestimmte Software- und Hardwareversionen beschränkt.

Die Informationen in diesem Dokument wurden von den Geräten in einer bestimmten Laborumgebung erstellt. Alle in diesem Dokument verwendeten Geräte haben mit einer leeren (Standard-)Konfiguration begonnen. Wenn Ihr Netzwerk in Betrieb ist, stellen Sie sicher, dass Sie die potenziellen Auswirkungen eines Befehls verstehen.

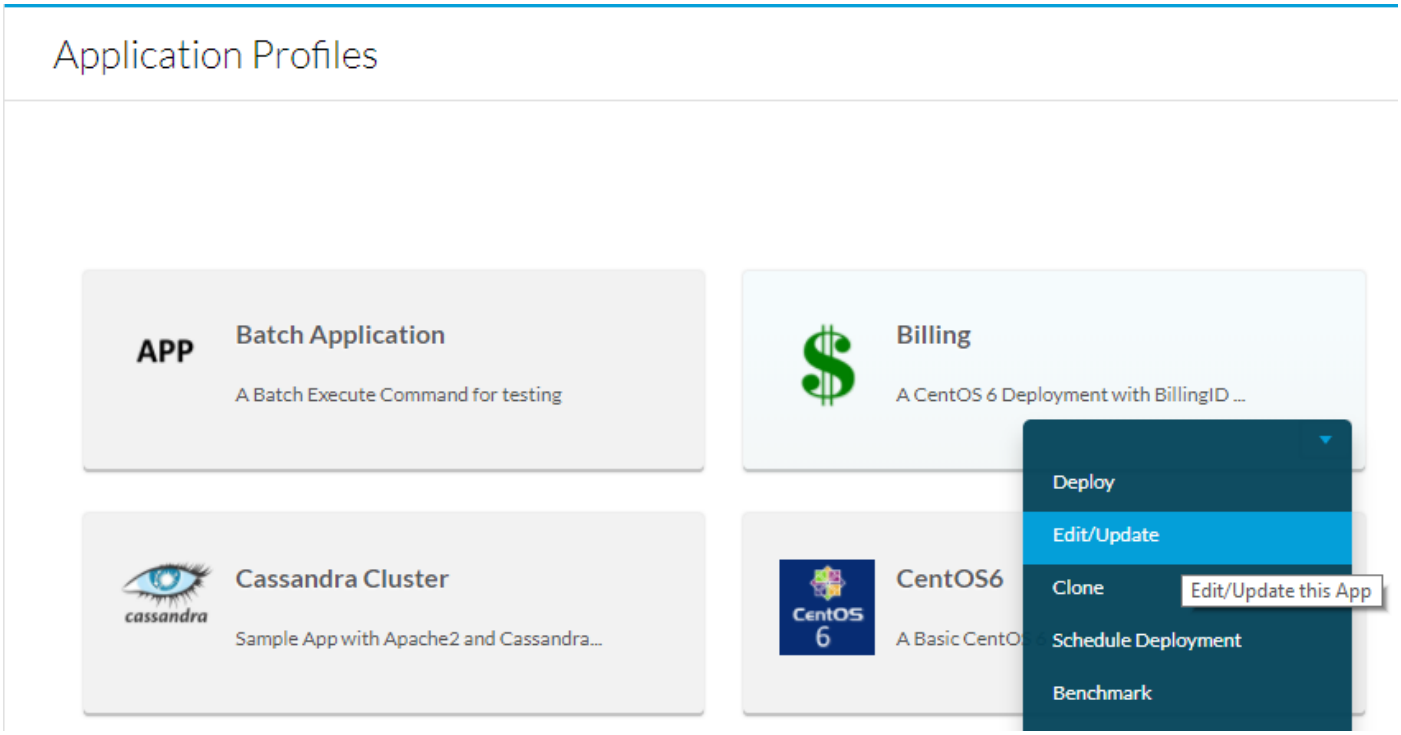
Hintergrundinformationen

CloudCenter bietet zwar standardmäßig einige Berichte, ermöglicht jedoch keine Berichterstellung auf der Grundlage benutzerdefinierter Filter. Um die Informationen direkt aus der Datenbank mithilfe von APIs in Verbindung mit Metadaten, die an die Jobs angefügt sind, abzurufen, können Sie benutzerdefinierte Berichte zulassen.

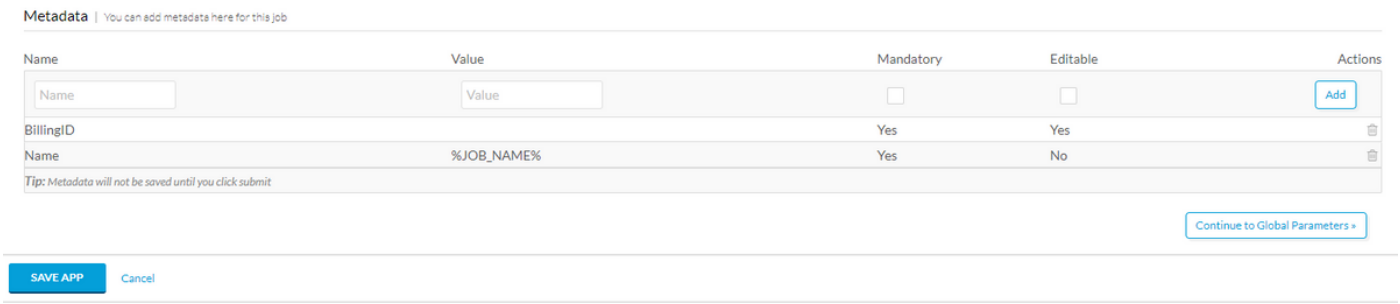
Einrichten der Metadaten

Metadaten müssen auf Anwendungsebene hinzugefügt werden, sodass jede Anwendung, die mit der Verwendung des benutzerdefinierten Berichts verfolgt werden muss, geändert werden muss.

Navigieren Sie dazu zu **Anwendungsprofile**, wählen Sie dann das Dropdown-Menü für die zu bearbeitende App aus, und wählen Sie dann **Bearbeiten/Aktualisieren** aus, wie im Bild gezeigt.



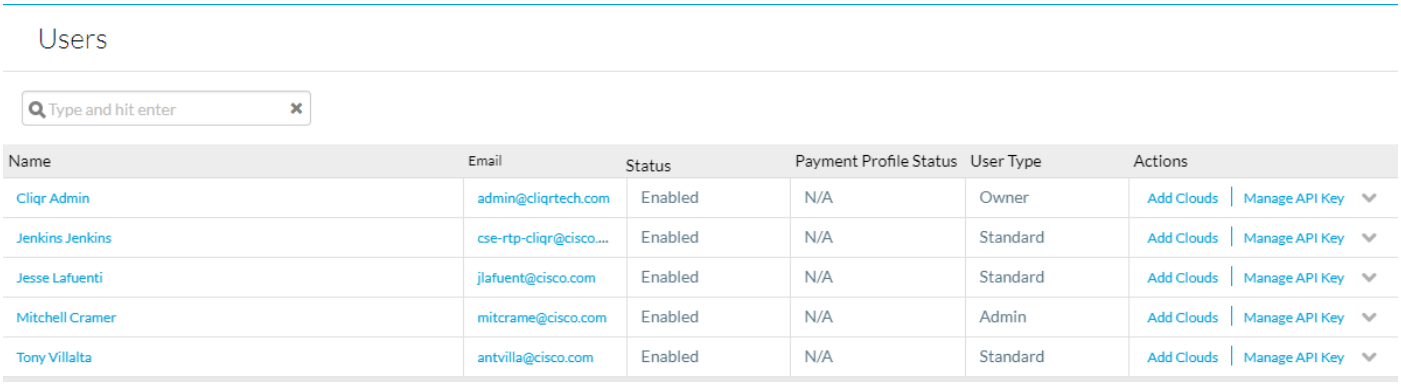
Scrollen Sie zum unteren Rand der **Basisinformationen**, und fügen Sie ein Metadata-Tag hinzu, z. B. **BillingID**, wenn diese Metadaten vom Benutzer ausgefüllt werden sollen, damit sie sowohl obligatorisch als auch bearbeitbar sind. Wenn es sich nur um ein Makro handelt, geben Sie den Standardwert ein und lassen Sie es nicht bearbeitbar. Nachdem Sie die Metadaten ausgefüllt haben, wählen Sie **Add** und dann **Save App** wie im Bild gezeigt aus.



Zusammenstellen von API-Schlüsseln

Für die Verarbeitung der API-Aufrufe sind Benutzername- und API-Schlüssel erforderlich. Diese Schlüssel bieten die gleiche Zugriffsstufe wie der Benutzer. Wenn also alle Benutzerbereitstellungen dem Bericht hinzugefügt werden sollen, wird empfohlen, die Admin-Schlüssel der Tenant-API abzurufen. Wenn mehrere Untertenants zusammen aufgezeichnet werden sollen, muss entweder der Root-Tenant auf alle Bereitstellungsumgebungen zugreifen oder die API-Schlüssel aller Untertenadministratoren müssen vorhanden sein.

Um die API-Schlüssel zu erhalten, navigieren Sie zu **Admin > Users > Manage API Key**, kopieren Sie den Benutzernamen und den Schlüssel für die erforderlichen Benutzer.



Erstellen des benutzerdefinierten Berichts

Stellen Sie vor dem Erstellen des Python-Skripts, das den Bericht erstellt, sicher, dass python und pip installiert sind. Führen Sie dann **pip install tabulate aus**, und tabulieren Sie eine Bibliothek, die die automatische Formatierung des Berichts behandelt.

Zwei Beispielberichte sind diesem Leitfaden beigefügt. Der erste sammelt einfach Informationen über alle Bereitstellungen und gibt sie dann in einer Tabelle aus. Die zweite verwendet die gleichen Informationen, um einen benutzerdefinierten Bericht unter Verwendung von BillingID-Metadaten zu erstellen. Dieses Skript wird detailliert als Leitfaden erläutert.

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

Datetime wird verwendet, um das Datum genau zu berechnen, so wird ein Bericht der letzten X Tage erstellt.

json wird verwendet, um die Analyse von Json-Daten zu unterstützen, die Ausgabe von API-Anrufen.

sys wird für Systemanrufe verwendet.

Anfragen werden verwendet, um die Erstellung von Webanfragen für API-Aufrufe zu vereinfachen.

Tabulator wird verwendet, um die Tabelle automatisch zu formatieren.

itemgetter wird als Iterator zum Sortieren einer 2D-Tabelle verwendet.

Dezimal wird verwendet, um die Kosten auf zwei Dezimalstellen zu runden.

```
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()
```

Dieser Teil wird verwendet, um den Befehlszeilenparameter der Anzahl der Tage zu analysieren.

Wenn keine Befehlszeilenparameter vorhanden sind (sys.argv ==1), erfolgt die Berichterstellung für alle Zeiten.

Wenn es einen Befehlszeilenparameter gibt, überprüfen Sie, ob es sich um eine Ganzzahl größer oder gleich 1 handelt, falls dieser bei dieser Anzahl von Tagen gemeldet wird, geben Sie andernfalls einen Fehler zurück.

Wenn mehr als ein Parameter vorhanden ist, wird ein Fehler ausgegeben.

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

Abteilungen ist die Liste, die die endgültige Ausgabe enthalten wird.

Benutzer ist eine Liste aller Benutzer, die die API-Anrufe tätigen. Wenn es mehrere UnterTenants gibt, wäre jeder Benutzer der Administrator eines anderen Untertentenants.

passwords ist eine Liste der Benutzer-API-Schlüssel, die Reihenfolge der Benutzer und Schlüssel muss identisch sein, damit der richtige Schlüssel verwendet werden.

```
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]])
```

für j in xrange(0,len(Users)): für Schleife, die durch jeden im vorherigen Code-Chunk definierten Benutzer durchlaufen wird, ist dies die Hauptschleife, die alle API-Aufrufe verarbeitet.

Jobs ist eine temporäre Liste, die verwendet wird, um die Informationen für Jobs zu speichern, während sie in der Liste sortiert werden.

r = request.get..... ist der erste API-Aufruf, dieser nennt alle Jobs, für weitere Informationen siehe List Jobs.

Die Ergebnisse werden dann im Einzelformat in **Daten** gespeichert.

für i in xrange(0,len(data["jobs"]): durchläuft alle Jobs, die vom vorherigen API-Aufruf zurückgegeben wurden.

Die Zeit für jeden Job wird aus dem Json abgerufen und in ein DataTime-Objekt konvertiert. Anschließend wird er mit dem eingegebenen Befehlszeilenparameter verglichen, um festzustellen, ob er innerhalb von Grenzen liegt.

Ist dies der Fall, werden diese Informationen aus dem Json der Jobliste hinzugefügt: **id, totalCost, status, name, Startzeit**. Nicht alle diese Informationen werden verwendet, und auch nicht alle Informationen, die zurückgegeben werden können. Listenjobs zeigt alle zurückgegebenen Informationen an, die auf dieselbe Weise hinzugefügt werden können.

Nachdem Sie alle von diesem Benutzer zurückgegebenen Jobs durchlaufen haben, wechseln Sie zu **ID in Jobs:** , der alle Aufträge durchläuft, die nach dem Überprüfen des Startdatums ausgeführt wurden.

q = request.get(..... ist der zweite API-Aufruf, dieser hier listet alle Informationen auf, die sich auf die Job-ID beziehen, die vom ersten API-Aufruf übernommen wurde. Weitere Informationen finden Sie unter Abrufen von Auftragsdetails.

Die json-Datei wird dann in **data2** gespeichert.

Die Kosten, die in der **ID[2]** gespeichert werden**,** werden auf zwei Dezimalstellen gerundet.

für i in xrange(0,len(data2["Metadaten"]): durchläuft alle Metadaten, die dem Job zugeordnet sind.

Wenn Metadaten mit der Bezeichnung **BillingID** vorhanden sind, werden sie in den Jobinformationen gespeichert.

hinzugefügt wird ein Flag, mit dem bestimmt wird, ob die **Rechnungsnummer** bereits der Liste der **Abteilungen** hinzugefügt wurde.

für i in xrange(0,len(Abteilungen)): durchläuft alle hinzugefügten Abteilungen.

Wenn dieser Job Teil einer Abteilung ist, die bereits existiert, wird die Anzahl der Jobs durch eine Iteration durchlaufen, und die Kosten werden zu den Gesamtkosten für diese Abteilung addiert.

Ist dies nicht der Fall, wird eine neue Zeile an Abteilungen angefügt, deren Job eine Zahl von 1 hat und deren Gesamtkosten den Kosten dieser einen Arbeit entsprechen.

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

Abteilungen = sorted(Abteilungen, key=itemgetter(1)) sortiert die Abteilungen nach der Anzahl der Jobs.

print(tabulate(abteilungen,headers=['Department','Number of Jobs', 'Total Cost'])) druckt eine Tabelle, die mit drei Headern tabellarisch erstellt wurde.

Zugehörige Informationen

- CloudCenter-API
- Technischer Support und Dokumentation - Cisco Systems