**Seismic Poster**

di Maria Pennicchi

**Corso di Virtual Network and Cloud Computing**

**Immagine che contiene testo, logo, Carattere, Elementi grafici

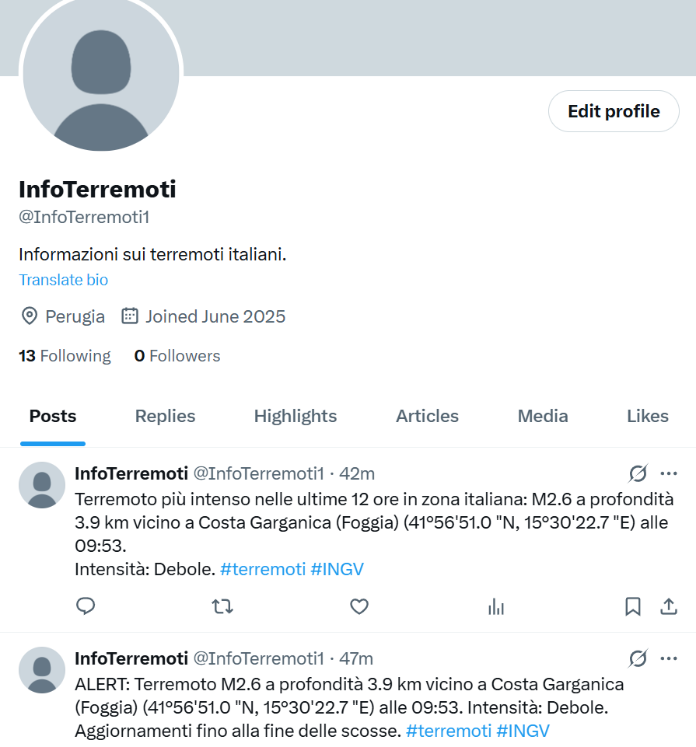
Il contenuto generato dall'IA potrebbe non essere corretto.**

Questo progetto si propone di realizzare un sistema serverless distribuito per il monitoraggio e la pubblicazione automatica di eventi sismici avvenuti in Italia. L’obiettivo principale è quello di raccogliere periodicamente dati da fonti ufficiali, nel nostro caso INGV [1], elaborarli in tempo reale e pubblicare messaggi informativi su X, piattaforma conosciuta precedentemente come Twitter, in modo completamente automatizzato.

La soluzione è implementata su un cluster Kubernetes tramite OpenFaaS, una piattaforma serverless che consente di eseguire funzioni su richiesta. Due funzioni principali sono state sviluppate in Python:

* L’handler di **poster-info**, che pubblica informazioni sul terremoto più intenso avvenuto nelle ultime 12 ore a intervalli regolari, precisamente alle 8 e alle 20 italiane di ogni giorno;
* L’handler di **poster-alert**, che pubblica informazioni per avvisare di scosse sismiche intense (con magnitudo superiore a 4.5). Questa funzione in particolare simula un event listener, in quanto è attivata da un cronjob ogni 15 minuti per verificare se ci sono state nuove scosse intense e pubblicarne degli aggiornamenti. Purtroppo, è stata necessaria formularla in questo modo in quanto le API di INGV non prevedono la pubblicazione di avvisi. Inizialmente, era stata pubblicata ogni 2 minuti e controllava i terremoti nei 2 minuti precedenti, ma dato che INGV API fornisce i dati sui terremoti con circa 11 minuti di ritardo, si è deciso di aumentare la finestra a 15 minuti per riuscire a intercettarli.

L’architettura Kubernetes si basa su CronJob per la chiamata periodica delle funzioni, attivate in OpenFaaS. Si usa inoltre un Secret Kubernetes per gestire le credenziali di accesso alle API di X Developer in modo sicuro.



Implementazione

**1)** Per prima cosa bisogna creare un account su X, io ho usato ‘@[InfoTerremoti1](https://x.com/InfoTerremoti1)’, per poi candidarsi per avere un account su [X Developer](https://developer.x.com/en) [2]. Si deve poi creare un progetto ‘Seismic\_Poster’, su cui poi verranno pubblicati i post. Poi è necessario attivare l’autenticazione OAuth 1.0 seguendo il modulo apposito trovabile nei settings del progetto e cambiare le impostazioni dell’account su ReadandWritePosts in modo tale da poter anche postare in automatico. A questo punto è fondamentale generare e salvare API Key, API Secret, Access Token e Access Token Secret, che ci permetteranno di autenticare le richieste HTTPS per postare via Python [3].

**2)** A questo punto possiamo spostarci sul cluster Kubernetes. Io per risparmiare tempo ho utilizzato le due macchine virtuali implementate durante i laboratori del corso [4], in cui avevamo già installato un cluster Kubernetes composto da un nodo master e un nodo worker tramite Kubespray. In queste macchine avevamo anche installato OpenFaaS, tramite arkade, faas-cli e docker.

**3)** Aprendo un terminale dal nodo master ci creiamo una cartella di lavoro seismic\_poster e vi entriamo

$ mkdir seismic\_poster

$ cd seismic\_poster/

**4)** A questo punto creiamo il namespace in cui lavoreremo in Kubernetes e applichiamo le modifiche

$ nano namespace.yaml

apiVersion: v1

kind: Namespace

metadata:

name: seismic-poster

$ kubectl apply -f namespace.yaml

**5)** Creiamo ora il Secret per tenere al sicuro le nostre credenziali di X e poterle richiamare. Dato che dovranno essere richiamate in OpenFaaS, il namespace sarà quello di default per le funzioni in OpenFaaS, ovvero openfaas-fn.

$ nano twitter-secrets.yaml

apiVersion: v1

kind: Secret

metadata:

name: twitter-secrets

namespace: openfaas-fn

type: Opaque #tipo generico

stringData:

X\_API\_KEY: “<codice>”

X\_API\_SECRET: “<codice>”

X\_ACCESS\_TOKEN: “<codice>”

X\_ACCESS\_SECRET: “<codice>”

$ kubectl apply -f twitter-secrets.yaml

**6)** Creiamo ora i due cronjob che ci serviranno a richiamare le due funzioni OpenFaaS. Prima creiamo cronjob-info, che richiamerà poster-info dal gateway di OpenFaaS. Osserviamo che il gateway di openfaas avrà un indirizzo DNS, solo nel cluster Kubernetes, così composto <nome-servizio>.<namespace>.svc.cluster.local, e la porta su cui è in ascolto è la 8080, verificabile tramite il comando

$ kubectl get svc -n openfaas

Scriviamo ora il cronjob\_info.yaml

$ nano cronjob\_info.yaml

apiVersion: batch/v1 #essendo un job inizia e finisce quindi serve batch

kind: CronJob

metadata:

name: cronjob-info

namespace: seismic-poster

spec:

schedule: "0 6,18 \* \* \*" #min, ore, giorno, mese, giorno settimana in utc, / indica il modulo

successfulJobsHistoryLimit: 3

failedJobsHistoryLimit: 1

jobTemplate:

spec:

backoffLimit: 1 #ritentare una volta

template:

spec:

restartPolicy: OnFailure

containers:

- name: trigger-info

image: curlimages/curl:7.85.0

command: ["sh", "-c"] #eseguo shell sh e comando di tipo stringa

args:

- >

curl --max-time 10 -s -X POST

<http://gateway.openfaas.svc.cluster.local:8080/function/poster-info>

#unisce in stringa multilinea, timeout 10s di attesa di risposta dal server, -s silenzioso, HTTP POST per attivare una funzione all’indirizzo interno al cluster di gateway di openfaas

$ kubectl apply -f cronjob\_info.yaml

Ora facciamo la stessa cosa con cronjob\_alert, che richiamerà poster-alert, sempre da OpenFaaS

$ nano cronjob\_alert.yaml

apiVersion: batch/v1

kind: CronJob

metadata:

name: cronjob-alert

namespace: seismic-poster

spec:

schedule: "\*/15 \* \* \* \*" #ogni 15 minuti

successfulJobsHistoryLimit: 3

failedJobsHistoryLimit: 1

jobTemplate:

spec:

backoffLimit: 1

template:

spec:

restartPolicy: OnFailure

containers:

- name: trigger-alert

image: curlimages/curl:7.85.0

command: ["sh", "-c"]

args:

- >

curl --max-time 10 -s -X POST

http://gateway.openfaas.svc.cluster.local:8080/function/poster-alert

$ kubectl apply -f cronjob\_alert.yaml

**7)** A questo punto si decide di procedere usando un template già disponibile nello store di OpenFaaS, perciò va scaricato in locale

$ faas-cli template store pull python3-flask

E poi vengono usati per creare le due funzioni serverless poster-info e poster-alert.

**8)** Iniziamo con poster-info.

$ faas-cli new poster-info --lang python3-flask-debian

E rinominiamo lo stack.yaml che viene generato in poster\_info.yaml

$ mv stack.yaml poster\_info.yaml

Dobbiamo ora modificare poster\_info.yaml aggiungendo il nostro username di dockerhub nel campo image prima del nome dell’immagine, e aggiungendo il riferimento al Secret che verrà usato dall’handler.py. Dato che stiamo testando in locale usiamo come gateway di OpenFaaS l’indirizzo di loopback della nostra macchina e la porta 8080 su cui faremo port-forwarding e da cui sarà accessibile questo servizio. Se avessimo fatto un deployment in cloud l’IP sarebbe dovuto essere quello del gateway esterno.

$ nano poster\_info.yaml

version: 1.0

provider:

name: openfaas

gateway: <http://127.0.0.1:8080> #indirizzo al quale posso accedere al gateway openfaas dall’esterno del cluster grazie al port forwarding

functions:

poster-info:

lang: python3-flask-debian

handler: ./poster-info

image: mariapennicchi/poster-info:latest

namespace: openfaas-fn

secrets:

- twitter-secrets

Nella directory poster-info dobbiamo aggiornare ora il file requirements.txt per specificare le dipendenze

$ cd poster-info/

$ nano requirements.txt

requests

datetime

requests\_oauthlib

E aggiungere il codice python nella funzione handler.py. Qui bisogna fare attenzione perché non servirà un main, ma una funzione handle() che verrà richiamata da OpenFaaS quando verrà invocate la funzione serverless poster-info.

$ nano handler.py

import requests  
import datetime

import time  
from requests\_oauthlib import OAuth1  
  
#Twitter API keys  
with open('/var/openfaas/secrets/X\_API\_KEY') as f:  
 API\_KEY = f.read().strip()  
with open('/var/openfaas/secrets/X\_API\_SECRET') as f:  
 API\_SECRET = f.read().strip()  
with open('/var/openfaas/secrets/X\_ACCESS\_TOKEN') as f:  
 ACCESS\_TOKEN = f.read().strip()  
with open('/var/openfaas/secrets/X\_ACCESS\_SECRET') as f:  
 ACCESS\_SECRET = f.read().strip()  
  
#INGV API endpoint  
INGV\_URL = "https://webservices.ingv.it/fdsnws/event/1/query"  
  
ITALY\_BOUNDS = {  
 "minlat": 35.0,  
 "maxlat": 47.5,  
 "minlon": 6.0,  
 "maxlon": 19.0,  
}  
  
def get\_danger\_level(mag):  
 if mag < 2.5:  
 return "Non avvertito"  
 elif mag < 3.5:  
 return "Debole"  
 elif mag < 5.0:  
 return "Moderato"  
 elif mag < 6.0:  
 return "Forte"  
 elif mag < 7.0:  
 return "Molto forte"  
 else:  
 return "Estremamente pericoloso"  
  
def convert\_latitude(lat):  
 direction = "N" if lat >= 0 else "S"  
 value = abs(lat)  
 degrees = int(value)  
 minutes\_float = (value - degrees) \* 60  
 minutes = int(minutes\_float)  
 seconds = (minutes\_float - minutes) \* 60  
 return f"{degrees}°{minutes}'{seconds:.1f} \"{direction}"  
  
def convert\_longitude(lon):  
 direction = "E" if lon >= 0 else "W"  
 value = abs(lon)  
 degrees = int(value)  
 minutes\_float = (value - degrees) \* 60  
 minutes = int(minutes\_float)  
 seconds = (minutes\_float - minutes) \* 60  
 return f"{degrees}°{minutes}'{seconds:.1f} \"{direction}"  
  
def decimal\_to\_dms(lat, lon):  
 lat\_dms = convert\_latitude(lat)  
 lon\_dms = convert\_longitude(lon)  
 return lat\_dms, lon\_dms  
  
def is\_location\_in\_italy(lat, lon):  
 return (  
 ITALY\_BOUNDS["minlat"] <= lat <= ITALY\_BOUNDS["maxlat"]  
 and ITALY\_BOUNDS["minlon"] <= lon <= ITALY\_BOUNDS["maxlon"]  
 )  
  
def get\_strongest\_quake():  
 now = datetime.datetime.utcnow()  
 start\_time = (now - datetime.timedelta(hours=12)).isoformat()  
  
 params = {  
 "starttime": start\_time,  
 "endtime": now.isoformat(),  
 "format": "geojson",  
 \*\*ITALY\_BOUNDS  
 }  
  
 r = requests.get(INGV\_URL, params=params, verify=False) #invia una richiesta HTTP GET  
 r.raise\_for\_status()  
 data = r.json() #formato di testo strutturato  
  
 if not data["features"]:  
 print("Non ci sono terremoti.", flush=True)  
 return None  
  
 valid\_quakes = []  
 for q in data["features"]:  
 try:  
 coords = q["geometry"]["coordinates"]  
 lat, lon = coords[1], coords[0]  
 if (is\_location\_in\_italy(lat, lon)):  
 valid\_quakes.append(q)  
 except Exception:  
 continue  
  
 if not valid\_quakes:  
 print("Non ci sono terremoti validi.", flush=True)  
 return None  
  
 # Ordina per magnitudo decrescente  
 valid\_quakes.sort(key=lambda f: f["properties"]["mag"], reverse=True)  
 quake = valid\_quakes[0]  
  
 props = quake["properties"]  
 coords = quake["geometry"]["coordinates"]  
 mag = props["mag"]  
 place = props["place"]  
 depth\_km = coords[2]  
 lat, lon = coords[1], coords[0]  
 lat\_dms, lon\_dms = decimal\_to\_dms(lat, lon)  
 utc\_dt = datetime.datetime.fromisoformat(props["time"]).replace(tzinfo=datetime.timezone.utc)  
 italy\_dt = utc\_dt + datetime.timedelta(hours=2)  
 time = italy\_dt.strftime("%H:%M")  
 danger = get\_danger\_level(mag)  
  
 return (f"Terremoto più intenso nelle ultime 12 ore in zona italiana: M{mag:.1f} a profondità {depth\_km:.1f} km vicino a {place} ({lat\_dms}, {lon\_dms}) alle {time}.\n"  
 f"Intensità: {danger}. #terremoti #INGV")  
  
def post\_to\_twitter(text):  
 url = "<https://api.twitter.com/2/tweets>" # Accetta richieste HTTP POST  
 auth = OAuth1(API\_KEY, API\_SECRET, ACCESS\_TOKEN, ACCESS\_SECRET) #cripta le credenziali per l’invio sicuro  
 headers = {  
 "Content-Type": "application/json"  
 } #tipo di contenuto della richiesta  
 payload = {"text": text}  
 resp = requests.post(url, json=payload, headers=headers, auth=auth)  
 resp.raise\_for\_status()  
 return resp.json()  
  
def handle(event\_body):  
 msg = get\_strongest\_quake()  
 try:  
 post\_to\_twitter(msg)  
 print("Tweet pubblicato", flush=True)  
 except requests.exceptions.HTTPError as e:  
 print("Errore HTTP:", e.response.status\_code, e.response.text)  
 raise  
 return "Fine funzione info."

$ cd ..

**9)** Ripetiamo il procedimento per poster-alert.

$ faas-cli new poster-alert --lang python3-flask-debian

$ mv stack.yaml poster\_alert.yaml

Modifichiamo ora poster\_alert.yaml aggiungendo il nostro username di dockerhub e il riferimento al Secret che verrà usato dall’handler.py.

$ nano poster\_alert.yaml

version: 1.0

provider:

name: openfaas

gateway: http://127.0.0.1:8080

functions:

poster-alert:

lang: python3-flask-debian

handler: ./poster-alert

image: mariapennicchi/poster-alert:latest

namespace: openfaas-fn

secrets:

- twitter-secrets

$ cd poster-alert/

$ nano requirements.txt

datetime

requests

requests\_oauthlib

$ nano handler.py

import requests  
from datetime import datetime, timezone, timedelta  
from requests\_oauthlib import OAuth1  
  
#Twitter API keys  
with open('/var/openfaas/secrets/X\_API\_KEY') as f:  
 API\_KEY = f.read().strip()  
with open('/var/openfaas/secrets/X\_API\_SECRET') as f:  
 API\_SECRET = f.read().strip()  
with open('/var/openfaas/secrets/X\_ACCESS\_TOKEN') as f:  
 ACCESS\_TOKEN = f.read().strip()  
with open('/var/openfaas/secrets/X\_ACCESS\_SECRET') as f:  
 ACCESS\_SECRET = f.read().strip()  
  
#INGV API endpoint  
INGV\_URL = "https://webservices.ingv.it/fdsnws/event/1/query"  
  
ITALY\_BOUNDS = {  
 "minlat": 35.0,  
 "maxlat": 47.5,  
 "minlon": 6.0,  
 "maxlon": 19.0,  
}  
  
def convert\_latitude(lat):  
 direction = "N" if lat >= 0 else "S"  
 value = abs(lat)  
 degrees = int(value)  
 minutes\_float = (value - degrees) \* 60  
 minutes = int(minutes\_float)  
 seconds = (minutes\_float - minutes) \* 60  
 return f"{degrees}°{minutes}'{seconds:.1f} \"{direction}"  
  
def convert\_longitude(lon):  
 direction = "E" if lon >= 0 else "W"  
 value = abs(lon)  
 degrees = int(value)  
 minutes\_float = (value - degrees) \* 60  
 minutes = int(minutes\_float)  
 seconds = (minutes\_float - minutes) \* 60  
 return f"{degrees}°{minutes}'{seconds:.1f} \"{direction}"  
  
def decimal\_to\_dms(lat, lon):  
 lat\_dms = convert\_latitude(lat)  
 lon\_dms = convert\_longitude(lon)  
 return lat\_dms, lon\_dms  
  
def is\_location\_in\_italy(lat, lon):  
 return (  
 ITALY\_BOUNDS["minlat"] <= lat <= ITALY\_BOUNDS["maxlat"]  
  
 and ITALY\_BOUNDS["minlon"] <= lon <= ITALY\_BOUNDS["maxlon"]  
 )  
  
def get\_danger\_level(mag):  
 if mag < 2.5:  
 return "Non avvertito"  
 elif mag < 3.5:  
 return "Debole"  
 elif mag < 5.0:  
 return "Moderato"  
 elif mag < 6.0:  
 return "Forte"  
 elif mag < 7.0:  
 return "Molto forte"  
 else:  
 return "Estremamente pericoloso"  
  
def get\_recent\_m4\_5\_quakes():  
 now = datetime.utcnow()  
 start\_time = (now - timedelta(minutes=15)).isoformat()  
  
 params = {  
 "starttime": start\_time,  
 "endtime": now.isoformat(),  
 "minlat": 35.0,  
 "maxlat": 47.5,  
 "minlon": 6.0,  
 "maxlon": 19.0,  
 "minmag": 4.5,   
 "format": "geojson",  
 }  
  
 r = requests.get(INGV\_URL, params=params, verify=False)  
  
 r.raise\_for\_status()  
 if not r.text.strip():  
 print("Nessuna risposta dal servizio INGV.", flush=True)  
 return []  
  
 try:  
 data = r.json()  
 except ValueError as e:  
 print("Risposta non in formato JSON valido:", r.text[:100], "...")  
 return []  
  
 return data.get("features", [])  
  
def post\_to\_twitter(text):  
 url = "https://api.twitter.com/2/tweets"  
 auth = OAuth1(API\_KEY, API\_SECRET, ACCESS\_TOKEN, ACCESS\_SECRET)  
 headers = {  
 "Content-Type": "application/json"  
 }  
 payload = {"text": text}  
 resp = requests.post(url, json=payload, headers=headers, auth=auth)  
 resp.raise\_for\_status()  
 return resp.json()  
  
def handle(event\_body):  
 recent\_quakes = get\_recent\_m4\_5\_quakes()  
   
 if recent\_quakes:  
 # Se ci sono nuove scosse M≥4.5  
 strongest = sorted(recent\_quakes, key=lambda q: q["properties"]["mag"], reverse=True)[0]  
 props = strongest["properties"]  
 coords = strongest["geometry"]["coordinates"]  
 place = props["place"]  
 mag = props["mag"]  
 depth = coords[2]  
 lat, lon = coords[1], coords[0]  
 lat\_dms, lon\_dms = decimal\_to\_dms(lat, lon)  
 utc\_dt = datetime.fromisoformat(props["time"]).replace(tzinfo=timezone.utc)  
 italy\_dt = utc\_dt + timedelta(hours=2)  
 time = italy\_dt.strftime("%H:%M")  
 danger = get\_danger\_level(mag)  
  
 msg = (  
 f"ALERT: Terremoto M{mag:.1f} a profondità {depth:.1f} km vicino a {place} ({lat\_dms}, {lon\_dms}) alle {time}. Intensità: {danger}.\n"  
 f"Aggiornamenti fino alla fine delle scosse. #terremoti #INGV"  
 )  
 try:  
 post\_to\_twitter(msg)  
 print("Tweet pubblicato.", flush=True)  
 except requests.exceptions.HTTPError as e:  
 print("Errore HTTP:", e.response.status\_code, e.response.text)  
 raise  
 else:  
 print("Non ci sono scosse intense.", flush=True)  
  
 return "Fine funzione alert."

$ cd ..

**10)** Ora è necessario fare il port-forward del gateway di OpenFaaS alla nostra porta localhost, infatti anche se il cluster gira in locale, Kubernetes usa indirizzi IP privati. Per accedere ai servizi dal nostro computer, usiamo il port forwarding, ovvero esponiamo il servizio gateway di OpenFaaS sulla porta 8080 del nostro localhost. Da un altro terminale quindi facciamo:

$ kubectl port-forward -n openfaas svc/gateway 8080:8080

Questo terminale va lasciato aperto in background per mantenere attivo il port forwarding.

**11)** Apriamo adesso un altro terminale ancora e eseguiamo il comando

$ echo $(kubectl get secret -n openfaas basic-auth -o jsonpath="{.data.basic-auth-password}" | base64 --decode)

Così da ottenere una password che possiamo inserire, insieme all’username “admin”, nella schermata web all’URL <http://127.0.0.1:8080> per accedere all’interfaccia web di OpenFaaS. Anche questo terminale lo possiamo mettere da parte così da avere velocemente sottomano le credenziali nel caso dovessimo ricaricare la pagina.

**12)** Tornando al nostro terminale dell’inizio connettiamoci a dockerhub con

$ docker login -u <username> -p <password>

**13)** Facciamo il build per creare le immagini docker delle funzioni poster-info e poster-alert e ne facciamo il push per caricarle su dockerhub. Dopodiché ne facciamo il deploy, con il quale faas-cli invia una richiesta HTTP POST al gateway OpenFaaS, che inoltra la richiesta a faas-netes. A questo punto faas-netes, dato che si tratta di un servizio sincrono, in modo del tutto trasparente, creerà un Deployment, un Pod contenente l’immagine, monterà il Secret e creerà un Service per accedervi. Iniziamo da poster-info

$ sudo faas-cli build -f poster\_info.yaml

$ faas-cli push -f poster\_info.yaml

$ faas-cli deploy -f poster\_info.yaml

Proseguiamo con poster-alert

$ sudo faas-cli build -f poster\_alert.yaml

$ faas-cli push -f poster\_alert.yaml

$ faas-cli deploy -f poster\_alert.yaml

**14)** Ora si può verificare da OpenFaaS web interface che le funzioni sono state correttamente caricate nel repository di OpenFaaS e che sono in stato di ready. Volendo verificare i cronjob attivati di recente si può eseguire

$ kubectl get jobs -n seismic-poster

E lo stesso vale per i pod corrispondenti

$ kubectl get pods -n seismic-poster

Per ottenere gli IP dei servizi esposti da OpenFaaS si può fare

$ kubectl get svc -n openfaas-fn

Per richiamare la funzione poster-info senza aspettare l’invocazione del cronjob si può fare

$ faas-cli invoke poster-info

E premere ctrl+D per evitare l’input, poi aspettare qualche secondo e ricaricare la pagina X per osservare il risultato.

O per poster-alert

$ faas-cli invoke poster-alert

E premere ctrl+D per evitare l’input, poi aspettare qualche secondo e leggere i log per osservare il risultato.

Per visualizzare l’output del codice python e per verificare eventuali errori nell’esecuzione si può usare per poster-info

$ faas-cli logs poster-info --gateway http://127.0.0.1:8080 --namespace openfaas-fn

O per poster-alert

$ faas-cli logs poster-alert --gateway http://127.0.0.1:8080 --namespace openfaas-fn  
  
Testare il funzionamento

**1)** Per testare il funzionamento della funzione poster-info, richiamata automaticamente dal cronjob-info ho modificato temporaneamente lo scheduler impostandolo a pochi minuti da quel momento. Dopo aver applicato le modifiche, ho controllato i log della funzione, dove avevo precedentemente inserito delle istruzioni print in handler.py, per verificare che l’intero flusso venisse eseguito correttamente. Ho infine confermato l’effettiva pubblicazione del post sull’account X associato, riscontrando che il contenuto generato era corretto.

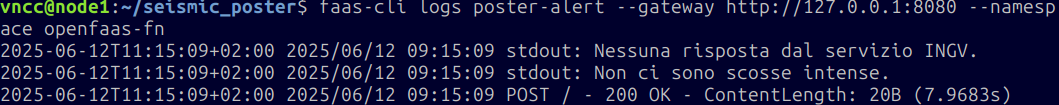
Immagine che contiene testo, schermata, Carattere

Il contenuto generato dall'IA potrebbe non essere corretto.

**2)** Per testare il funzionamento di poster-alert, invece, dato che per fortuna non ci sono stati terremoti forti in questi giorni, nel file handler.py ho abbassato temporaneamente la soglia minima di magnitudo a 1.0 e ampliato la finestra di tempo considerata a 90 minuti, perché ho verificato dal sito [INGV](https://terremoti.ingv.it/?l=1088&k=) che questo mi permetteva di avere almeno una scossa segnalata dalle API INGV in quell’intervallo e con quelle caratteristiche. Ho poi aggiornato lo scheduler di cronjob-alert per attivare la funzione nei minuti successivi e, dopo aver applicato le modifiche, ho controllato i log e l’account X per assicurarmi che il post fosse stato generato e pubblicato correttamente.



Ho poi ripristinato i parametri originali della query: soglia minima di magnitudo a 4.5, intervallo temporale di 15 minuti, e ho riportato la pianificazione del cronjob-alert alla sua frequenza periodica. Dopo aver rifatto il build, push e deploy della funzione e l’apply del cronjob-alert, ho atteso la successiva esecuzione automatica di quest’ultimo. Verificando i log, ho osservato che, in assenza di scosse significative, la funzione non pubblicava alcun tweet, ma stampava correttamente il messaggio previsto per il ramo condizionale che gestisce l’assenza di eventi.



Fonti

[1] Istituto Nazionale di Geofisica e Vulcanologia. “Portale Terremoti - Mappa Interattiva.” 2025. <https://terremoti.ingv.it/>. Accesso 11 giugno 2025.

[2] X Corp. “X Developer Platform.” 2025. <https://developer.x.com/en/>. Accesso 11 giugno 2025.

[3] X Corp. “X API: Introduction.” 2025. <https://docs.x.com/x-api/introduction>. Accesso 11 giugno 2025.

[4] Reali, Gianluca. “Kubernetes Journey.” 2025. <https://unistudium.unipg.it/unistudium/mod/resource/view.php?id=601534>. Accesso 11 giugno 2025.