TROPOSPHERE: UN TOOL PER CREARE, GESTIRE E MANTENERE UNA INFRASTRUTTURA AWS BASATA SU CLOUDFORMATION.

AWS CloudFormation | DevOps | Python | Software as a Service (SaaS)

beSharp | 7 Febbraio 2020

La gran parte delle moderne applicazioni web SaaS sono sviluppate e deployate sull'infrastruttura Cloud messa a disposizione da un Cloud Provider. In particolare Amazon Web Service è stato il primo vero Cloud provider e ha saputo conquistare e mantenere la guida di questo settore di mercato grazie alla qualità e alla flessibilità dei servizi offerti.

Le infrastrutture create su AWS stanno diventando sempre più grandi per sfruttare al massimo i vantaggi dei nuovi servizi rilasciati regolarmente da AWS. Il numero di servizi managed offerti direttamente da AWS è enorme e cresce di anno in anno.

L'utilizzo di servizi managed da AWS al posto, per esempio, di soluzioni custom implementate su macchine virtuali EC2 garantisce una notevole diminuzione dei costi di setup e manutenzione dell'infrastruttura date che Amazon si fa carico del deployment, dell'ottimizzazione, della manutenzione e anche della sicurezza di tutti i servizi offerti.

Inoltre la maggior parte dei servizi AWS è progettata per essere in alta affidabilità senza la necessità di configurazioni aggiuntive particolari, consentendo agli utilizzatori di evitare un altro grosso sforzo di configurazione e test.

L'utilizzo dei servizi managed da AWS come componenti base consente agli sviluppatori di aggredire con successo e in tempi rapidi un gran numero di problemi e di sviluppare praticamente ogni tipo di applicazione. Per esempio per una tipica applicazione web serverless moderna è possibile usare Amazon Cognito per l'autenticazione, DynamoDB come database, SNS/SES rispettivamente per notifiche push e mail, S3/Cloudfront per servire il frontend e SQS per le code applicative interne.

Tuttavia la maggior parte delle applicazione ha un livello di complessità molto maggiore di questo (sono spesso richiesti modelli di Machine Learning, datalakes, connessioni vpn verso altri

networks/servizi, tipi diversi di database, sistemi di batch processing etc.) e quindi il numero di servizi e risorse dell'infrastruttura AWS sale rapidamente a numeri molto elevati (spesso centinaia di risorse create) e le infrastrutture risultanti sono così grandi e complesse che non possono più essere create e mantenute 'a mano'.

Infatti molto spesso modifiche ad un solo componente (per esempio un security group, un ruolo o una routing table) possono dar luogo ad effetti collaterali imprevisti rilevanti che possono bloccare vari servizi e potenzialmente l'intera applicazione.

In questi casi la gestione dell'infrastruttura tramite IaC (Infrastructure as Code) da un grosso aiuto. Tramite IaC è infatti possibile descrivere l'intera infrastruttura AWS scrivendo codice, quindi è possibile versionarlo usando Git come per ogni altro progetto software. Quando il codice IaC viene eseguito andrà a creare o ad aggiornare l'infrastruttura per portarla ad uno stato identico a quello descritto nel codice! Se in un secondo momento dovesse sorgere la necessità di modificare l'infrastruttura sarà sufficiente aggiornare il codice IaC, committare la modifica ed eseguire nuovamente il codice. In caso di problemi imprevisti è perciò possibile effettuare il rollback di tutta l'infrastruttura all'ultima versione funzionante.

Se ti sembra tutto troppo bello per essere vero probabilmente hai ragione! Ogni livello di astrazione che andiamo ad aggiungere ad un progetto software ha le sue problematiche e lo IaC non fa eccezione. Il primo problema che uno sviluppatore tipicamente incontra approcciandosi allo Iac è la scelta dello strumento, infatti vi sono due framework IaC principali al momento disponibile per AWS: Terraform and CloudFormation. Inizialmente abbiamo sperimentato con terraform ma sono emersi alcuni problemi significativi:

- Terraform usa un suo linguaggio custom che ha una sintassi molto limitata, non è possibile per esempio scrivere cicli e loops.
- Occasionalmente Terraform non aspetta la creazione di una risorsa in modo corretto, questo genera a cascata errori nella creazione di risorse dipendenti da quella in fase di creazione
- Due sviluppatori possono eseguire uno script terraform allo stesso momento dato che l'esecuzione è in locale. Questo ovviamente genera errori e possibili inconsistenze nell'infrastruttura creata. Se si vuole utilizzare Terraform è essenziale rendere possibile l'esecuzione del codice solo tramite una pipeline di CI/CID
- I Rollbacks sono spesso non gestiti correttamente
- Terraform viene sempre eseguito su una macchina/container singola, perciò può essere bloccato da guasti hardware o di networking lasciando l'infrastruttura in uno stato inconsistente.

L'unico vero use case di Terraform è perciò quello che prevede la creazione coordinata di risorse su più Cloud Providers (e.g. Azure, Google Cloud Platform e AWS).

CloudFormation invece è a sua volta un servizio totalmente gestito da AWS: l'utilizzatore deve semplicemente caricare su S3 un file YAML o JSON contenente la descrizione dell'infrastruttura da ottenere e Cloudformation si prenderà carico di eseguirlo in modo sicuro e stateful. I Rollback in caso di errore sono supportati nativamente ed è anche possibile eseguire dei dry runs del template creando dei Change Set al fine di capire quali risorse andranno ad essere effettivamente modificate durante l'esecuzione vera e propria del template. In generale l'esecuzione di un template

Cloudformation è significativamente meno prona ad errori di quella di un template Terraform anche grazie al fatto che il servizio è nativo AWS.

Tuttavia anche CloudFormation presenta alcune alcune criticità: i file YAML sono spesso molto lunghi e difficili sia da scrivere che da debuggare, inoltre, come nel caso di Terraform i cicli e la logica avanzata non sono supportati. Dividere un progetto in più moduli tramite i Nested Stacks è essenziale per motivi di manutenibilità del codice e per evitare di incorrere nel limite di 200 risorse per file, tuttavia i Nested Stack sono difficili da usare insieme ai Change Sets. Il prossimo passo nella strada verso una migliore esperienza IaC è perciò quello di generare i file YAML tramite un linguaggio di programmazione di alto livello come Python!

Anche qui abbiamo due possibilità: AWS CDK e Troposphere. AWS CDK è sviluppato da AWS, appena rilasciato e molto potente: con pochi comandi è possibile creare infrastrutture piuttosto complesse. Tuttavia il suo essere di alto livello è anche il suo più grande difetto: alcune operazioni di basso livello diventano difficili da implementare e lo YAML generato è di difficile comprensione per un essere umano dato che i nomi logici delle risorse create sono generati direttamente da CDK.

Al contrario Troposphere ha nella semplicità la sua più grande forza: si tratta di un Python DSL che va a mappare le entità di Cloudformation in oggetti Python (e viceversa!).

Questo paradigma ci fornisce esattamente ciò che stavamo cercando: un modo semplice per creare template cloudformation che non solo fanno ciò che vogliamo, ma che vengono generati esattamente come li vogliamo da un linguaggio di alto livello con tutti i costrutti di logica avanzati così utili in molteplici situazioni. Inoltre l'IDE di Python di nostra scelta ci aiuterà a risolvere errori e inconsistenze ancora prima di eseguire o validare lo YAML template e la compilazione stessa del codice da Python a YAML andrà in errore in caso di inconsistenze logiche rendendo molto più rapido il debugging.

Presentiamo qui un semplice esempio dell'uso di questo approccio per mostrarne potenza e versatilià: andremo a creare tramite Troposphere una VPC con 3 subnet, una per availability zone e la loro route table

Per prima cosa diamo però uno sguardo a come si presenterà il cloudformation YAML template per questa semplice infrastruttura:

```
Description: AWS CloudFormation Template to create a VPC
Parameters:
  SftpCidr:
    Description: SftpCidr
    Type: String
Resources:
  SftpVpc:
    Properties:
      CidrBlock: !Ref 'SftpCidr'
      EnableDnsHostnames: 'true'
      EnableDnsSupport: 'true'
    Type: AWS::EC2::VPC
  RouteTablePrivate:
    Properties:
```

```
      VpcId: !Ref 'SftpVpc'
    Type: AWS::EC2::RouteTable
  PrivateSubnet1:
    Properties:
      AvailabilityZone: !Select
        - 0
        - !GetAZs
          Ref: AWS::Region
      CidrBlock: !Select
        - 4
        - !Cidr
          - !GetAtt 'SftpVpc.CidrBlock'
          - 16
          - 8
      MapPublicIpOnLaunch: 'false'
      VpcId: !Ref 'SftpVpc'
    Type: AWS::EC2::Subnet
  PrivateSubnet2:
    Properties:
      AvailabilityZone: !Select
        - 1
        - !GetAZs
          Ref: AWS::Region
      CidrBlock: !Select
        - 5
        - !Cidr
          - !GetAtt 'SftpVpc.CidrBlock'
          - 16
          - 8
      MapPublicIpOnLaunch: 'false'
      VpcId: !Ref 'SftpVpc'
    Type: AWS::EC2::Subnet
  PrivateSubnet3:
    Properties:
      AvailabilityZone: !Select
        - 2
        - !GetAZs
          Ref: AWS::Region
      CidrBlock: !Select
        - 6
        - !Cidr
          - !GetAtt 'SftpVpc.CidrBlock'
          - 16
          - 8
      MapPublicIpOnLaunch: 'false'
      VpcId: !Ref 'SftpVpc'
    Type: AWS::EC2::Subnet
  SubnetPrivateToRouteTableAttachment1:
    Properties:
      RouteTableId: !Ref 'RouteTablePrivate'
      SubnetId: !Ref 'PrivateSubnet1'
    Type: AWS::EC2::SubnetRouteTableAssociation
  SubnetPrivateToRouteTableAttachment2:
    Properties:
      RouteTableId: !Ref 'RouteTablePrivate'
      SubnetId: !Ref 'PrivateSubnet2'
    Type: AWS::EC2::SubnetRouteTableAssociation
  SubnetPrivateToRouteTableAttachment3:
    Properties:
      RouteTableId: !Ref 'RouteTablePrivate'
```

```
      SubnetId: !Ref 'PrivateSubnet3'
    Type: AWS::EC2::SubnetRouteTableAssociation
```

Si può immediatamente notare che il codice YAML sebbene sia stato generato in modo programmatico è immediatamente leggibile e comprensibile. Buona parte del codice di questo esempio è in realtà duplicato perché stiamo andando a creare 3 subnets con le stesse caratteristiche e associazioni alla stessa routing table, ma con nomi logici e CIDR diversi.

Il codice Troposphere usato per generare il template appena visto è il seguente:

```
import troposphere.ec2 as vpc

template = Template()
template.set_description("AWS CloudFormation Template to create a VPC")

sftp_cidr = template.add_parameter(
        Parameter('SftpCidr', Type='String', Description='SftpCidr')
    )

vpc_sftp = template.add_resource(vpc.VPC(
        'SftpVpc',
        CidrBlock=Ref(sftp_cidr),
        EnableDnsSupport=True,
        EnableDnsHostnames=True,
    ))

private_subnet_route_table = template.add_resource(vpc.RouteTable(
        'RouteTablePrivate',
        VpcId=Ref(vpc_sftp)
    ))

for ii in range(3):
    private_subnet = template.add_resource(vpc.Subnet(
        'PrivateSubnet' + str(ii + 1),
        VpcId=Ref(vpc_sftp),
        MapPublicIpOnLaunch=False,
        AvailabilityZone=Select(ii, GetAZs(Ref(AWS_REGION))),
        CidrBlock=Select(ii + 4, Cidr(GetAtt(vpc_sftp, 'CidrBlock'), 16, 8))
    ))
    private_subnet_attachment = template.add_resource(vpc.SubnetRouteTableAssociation(
        'SubnetPrivateToRouteTableAttachment' + str(ii + 1),
        SubnetId=Ref(private_subnet),
        RouteTableId=Ref(private_subnet_route_table)
    ))

print(template.to_yaml())
```

L'esecuzione di questo script dopo aver installato Troposphere (pip install troposphere) manderà in output il template YAML. Come si può vedere il codice python è di gran lunga più compatto del template YAML e molto più semplice da leggere. Oltretutto, dato che Troposphere mappa anche tutte le funzioni native di CloudFormation (Ref, Join, GettAtt etc.) non abbiamo nemmeno bisogno di imparare nulla di nuovo: ogni template di cloudformation esistente può essere rapidamente convertito in un programma python.

A differenza di quanto si fa in CloudFormation con troposphere possiamo associare le variabili risorse a variabili Python e usare queste ultime come riferimento alla risorsa da passare come parametro alle funzioni Ref e GettAtt al posto dei nomi logici delle risorse: nell'esempio sopra abbiamo referenziato la subnet privata con

```
Ref(private_subnet_route_table), non con  Ref('RouteTablePrivate').
```

Questo è un grosso vantaggio in quanto non dobbiamo più ricordare i nomi delle risorse logiche definite nel template: l'IDE ci aiuterà a trovarle e ci avvertirà di eventuali inconsistenze.

Troposphere può anche gestire senza problemi i nested stack o altri tipi di architetture multistack attraverso il tool di automation Sceptre.

Invece di utilizzare Sceptre qui in beSharp abbiamo tuttavia deciso di creare un nostro script di deploy customizzato in modo da poter gestire con la massima versatilità questo step. L'utilizzo di uno script custom consente infatti di eseguire un CloudFormation drift change check prima di applicare ogni cambiamento all'infrastruttura e di valutare il ChangeSet per ognuno dei nested stacks prima di eseguire il template Cloudformation.

Troposphere è anche in grado di gestire il flusso inverso: a partire da uno YAML template trasformarlo in classi Python:

```
from cfn_tools import load_yaml
from troposphere import TemplateGenerator

template = TemplateGenerator(load_yaml(
            app_config.cloudformation.meta.client.get_template(
                StackName='MyStack')['TemplateBody']
        ))
```

Questo è molto utile in situazioni nelle quali vi sia la necessità di aggiornare in modo dinamico l'infrastruttura.

Per concludere l'utilizzo di Troposphere è un sistema molto comodo per cogliere tutti i vantaggi di CloudFormation con il livello di astrazione di un linguaggio di alto livello e semplifica notevolmente lo sviluppo ed il rilascio di codice Cloudformation.

Se ti interessa questo topic non esitare a scriverci nei commenti o a contattarci direttamente per ulteriori info!

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