PaaS su AWS: come implementarlo nel modo migliore – Parte II

14 Ottobre 2022 - 5 min. read

Amazon EC2 *Amazon RDS* *Amazon S3* *AWS CodeBuild* *AWS CodeDeploy*
AWS CodePipeline *CI/CD* *Continuous Delivery* *GitLab* *Platform-as-a-Service (PaaS)*
Virtual Host

Bentornati nella nostra mini-serie dedicata al Platform-as-a-Services su AWS.

Nella prima parte, abbiamo percorso i punti chiave di una implementazione ottimale introducendo i principali aspetti da considerare nello sviluppo di un prodotto PaaS. In questo articolo, esamineremo invece la questione dal punto di vista dello stack infrastrutturale analizzando i servizi condivisi alla base del funzionamento di una **Web Server vending machine**, il nostro obiettivo finale.

Partiamo dai repository.

Nel repository che analizzeremo, come accennato nel precedente articolo, troviamo i vari stack per la creazione di:

- Servizi per l'intercettazione dei **push su GitLab**:
  - API Gateway per accettare le chiamate dei webhook di GitLab
  - Lambda per creare un file di configurazione con i dati del commit effettuato
  - CodeBuild Job per effettuare il pull del reporitory e l'upload su S3

- **Ruoli IAM** dedicati per environment:

    - Ruolo per l'utilizzo della pipeline infrastrutturale
    - Instance profile per le istanze EC2
    - Ruolo per l'utilizzo della pipeline software
    - Ruolo per il deploy delle risorse legate alle istanze EC2, quali AutoScaling Group, Application Load Balancer, ...

- **VPC** per environment:
    - CIDR /16
    - 9 subnet:
        - 3 Private
        - 3 Natted
        - 3 Public

    - Istanze NAT

- **Chiave KMS** per la cifratura di ogni oggetto e servizio per environment
- **Bucket S3** per la gestione dei file utilizzati dalle pipeline (ad esempio gli artifact) suddiviso per ogni environment
- **Application load balancer** per ciascun environment:
    - Listener su porta 80 con redirect automatico sulla porta 443 in https
    - Listener su porta 443 con errore 503 in caso di non riscontro tra le regole

VPC

La VPC è composta da **9 subnet**, 3 per ogni Avaliability Zone, in maniera tale da rendere l'infrastruttura altamente disponibile. Le subnet sono suddivise in:

- PUBBLICHE, utilizzate per tutti i servizi che devono essere raggiunti dall'esterno (come l'ALB) o se si vuole esporre direttamente un'istanza EC2 su internet assegnandole un indirizzo IP pubblico dedicato.
- NATTATE, utilizzate per tutti i servizi che necessitano di accedere ad internet, ma che non devono essere raggiunti dall'esterno; come suggerito dal nome, le istanze che verranno create all'interno di queste subnet potranno accedere ad internet

tramite dei NAT gateway che risiedono nelle relative subnet pubbliche. Nel nostro caso, abbiamo scelto di optare per le 3 istanze (una per AZ) solo per la VPC di produzione, mentre per gli altri ambienti manterremo una sola istanza.

- PRIVATE, utilizzate per tutti i servizi che non necessitano di accesso ad internet come database RDS.

Il costrutto VPC che ci mette a disposizione AWS CDK ha una gestione dei CIDR assegnati alle subnet che rende impossibile effettuare supernetting, privandoci quindi della possibilità di raggruppare le subnet con netmask piu piccole. Abbiamo quindi deciso di utilizzare questo costrutto, assicurandoci però di sovrascrivere i vari CIDR prima del deploy tramite questa porzione di codice:

```
myVpc.privateSubnets.forEach((subnet, index) => {
let cidr = `${startSubnetsCidr}.${firstPrivateCidr + index}.${endSubn
etsCidr}`
const cfnSubnet = subnet.node.defaultChild as aws_ec2.CfnSubnet;
cfnSubnet.addPropertyOverride('CidrBlock', `${cidr}`);
let name =  `${configFile.projectName}-${process.env.ENVIRONMENT}-nat
ted-${subnet.availabilityZone.replace(/^\w+\-\w+\-\d/,'')}`;
let subName =  `Subnet-Natted-${subnet.availabilityZone.replace(/^\w+
\-\w+\-\d/,'').toUpperCase()}-${process.env.ENVIRONMENT}-Name`;
let subId =  `Subnet-Natted-${subnet.availabilityZone.replace(/^\w+\-
\w+\-\d/,'').toUpperCase()}-${process.env.ENVIRONMENT}-ID`;
let subCidr =  `Subnet-Natted-${subnet.availabilityZone.replace(/^\w+
\-\w+\-\d/,'').toUpperCase()}-${process.env.ENVIRONMENT}-CIDR`;
cdk.Aspects.of(subnet).add(
new cdk.Tag(
'Name',
Name
)
)
})
```

Rilasciata la VPC possiamo deployare al suo interno tutte le risorse necessarie per far funzionare la vending machine come ad esempio gli **Application Load Balancer** nelle

subnet pubbliche, i **Web Server** nelle subnet nattate, e i **database** dedicati ai Web Server nelle subnet private.

Affronteremo, nel prossimo articolo, la creazione di queste risorse.

Bucket Amazon S3

Il bucket S3 creato da questo stack viene utilizzato per stoccare i log, gli artifact e il risultato dei git push su GitLab; vengono inoltre assegnati relativi permessi per i ruoli IAM garantendo full access al bucket, e vengono create le removal policy per i log stoccati:

```
const myLifeCycleLogsRule: aws_s3.LifecycleRule = {
        id: `logs-cleared`,
enabled: true,
prefix: `*-${process.env.ENVIRONMENT}-log`,
expiration: cdk.Duration.days(1)
}
```

Per poter utilizzare il bucket S3 come sorgente della pipeline occorre attivare il servizio **CloudTrail** per garantire la possibilità di intercettare gli eventi:

```
const myTrail = new aws_cloudtrail.Trail(this, `CloudTrail-${process.
env.ENVIRONMENT}`, {
trailName: `trail-${process.env.ENVIRONMENT}`,
sendToCloudWatchLogs: true,
bucket: myGlobalBucketS3,
encryptionKey: myKms,
cloudWatchLogGroup: new aws_logs.LogGroup(this, `Logs-${upperEnvironm
ent}`, {
logGroupName: `logs-${process.env.ENVIRONMENT}`,
retention: aws_logs.RetentionDays.THREE_DAYS,
removalPolicy: RemovalPolicy.DESTROY
}),
cloudWatchLogsRetention: aws_logs.RetentionDays.THREE_DAYS,
s3KeyPrefix: `logs-${process.env.ENVIRONMENT}`,
```

```
isMultiRegionTrail: false
});
```

Ma questo non basta. Per far sì che la pipeline venga invocata all'inserimento di un nuovo file all'interno del bucket S3 è necessario configurare un **evento di notifica** su CloudTrail che stia in ascolto di operazioni di scrittura all'interno bucket S3:

```
myTrail.addS3EventSelector([{
        bucket: myGlobalBucketS3,
        objectPrefix: `software/`,
        }], {
        readWriteType: aws_cloudtrail.ReadWriteType.WRITE_ONLY,
})
myTrail.addS3EventSelector([{
        bucket: myGlobalBucketS3,
        objectPrefix: `infrastructure/`,
        }], {
        readWriteType: aws_cloudtrail.ReadWriteType.WRITE_ONLY,
})
```

KMS Key

Per garantire la cifratura dei dati su S3, su CloudTrail, e nel database, abbiamo creato una **chiave KMS customer managed**. A questa chiave abbiamo successivamente assegnato una policy che permette alle entità che devono operare sui servizi cifrati di poterla utilizzare:

```
myKms.addToResourcePolicy( new iam.PolicyStatement({
sid: "Allow principals in the account to decrypt log files",
actions: [
"kms:Decrypt",
"kms:ReEncryptFrom"
],
principals: [ new iam.AccountPrincipal(`${process.env.CDK_DEFAULT_ACC
OUNT}`) ],
resources: [
```

```
`arn:aws:kms:${process.env.CDK_DEFAULT_REGION}:${process.env.CDK_DEFA
ULT_ACCOUNT}:key/*`,
],
conditions: {
"StringLike": {
"kms:EncryptionContext:aws:cloudtrail:arn": "arn:aws:cloudtrail:*:*:t
rail/*"
},
"StringEquals": {
"kms:CallerAccount": `${process.env.CDK_DEFAULT_ACCOUNT}`
}
}
}));
```

Application Load Balancer

Questo ALB gestirà gli accessi ai nostri servizi indirizzandoli in automatico dalla porta 80 in HTTP alla porta 443 in HTTPS:

```
myAppLoadBalancer.addListener(`App-80-Listener`, {
port: 80,
defaultAction: elbv2.ListenerAction.redirect({
permanent: true,
port: '443',
protocol: 'HTTPS',
})
})
myAppLoadBalancer.addListener(`App-443-Listener`, {
port: 443,
defaultAction: elbv2.ListenerAction.fixedResponse(503, {
contentType: `text/plain`,
messageBody: 'host not found',
})
})
```

Per porter gestire le richieste effettuate in https sulla porta 443 è necessario che sia associato al relativo listener un certificato facilmente configurabile tramite il servizio AWS Certificate Manager, che oltre alla creazione dei certificati permette anche l'aggiornamento automatico degli stessi.

Conclusione

Le risorse configurate all'interno di questo repository costituiscono le fondamenta per l'intera soluzione.

Nella prossima puntata analizzeremo lo stack applicativo dedicato a ciascun cliente che utilizza i servizi che abbiamo visto oggi.

Per avere una soluzione solida dal punto di vista di sicurezza e scalabilità è necessario che ciò che la mantiene in piedi sia altrettanto affidabile: per questo motivo ci siamo appoggiati unicamente a servizi gestiti da AWS, riducendo quindi l'effort di amministrazione e monitoring.

Ci vediamo tra 14 giorni per l'ultimo capitolo!

Antonio Callegari

DevOps Engineer @ beSharp. Nasco sistemista "classico" innamorato di hardware, ma passo volentieri al lato oscuro: il Cloud! Preferisco sempre le cose fatte a mano, ma non disdegno un po' di sana automazione (se fatta con criterio...)Nel tempo libero allestisco e calco palchi e mi dedico alla mia famiglia

---