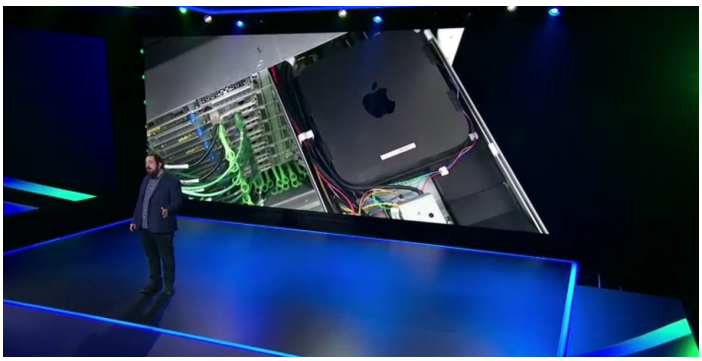
NUOVE ISTANZE AWS EC2 MAC: CI/CD COME BANCO DI PROVA

Amazon EC2 | Angular | AWS EC2 Mac | Electron

Eric Villa | 8 Gennaio 2021

Questo articolo è stato originariamente pubblicato su Hashnode.

Durante l'ultimo AWS re:Invent, AWS ha fatto uno degli annunci più discussi, e questo — almeno sulla carta — lascia spazio a nuovi possibili scenari di sviluppo: le istanze AWS EC2 Mac!
Non preoccupatevi, entreremo nel dettaglio di uno di questi scenari a breve nell'articolo, ma per ora, permettetemi di introdurvi questo nuovo tipo di Istanza.
Le istanze Amazon EC2 Mac hanno un nome che richiama gli anni 80, un fatto questo, che le rende sicuramente "attraenti" per quelli di voi un pò più anziani di me 🙂 .
Il loro nome è istanze **mac1.metal**.
Parlando invece di hardware, le Istanze Amazon EC2 Mac sono supportate da host di tipo Mac mini, dipendenti da controller AWS Nitro per connettersi ai servizi e all'infrastruttura di rete di AWS. Il punto interessante è che le istanze Mac sono interfacciate al sistema Nitro mediante tecnologia Thunderbolt 3. Ho volutamente utilizzato il termine "host" per sottolineare come non stiamo avendo a che fare con Macchine Virtuali, ma con veri e propri Host Dedicati; ogniqualvolta decido di avviare una Istanza Amazon EC2 Mac, AWS mi mette a disposizione concretamente un Host Mac mini dedicato ai miei scopi.



Peter Desantis, Senior Vice President of AWS Infrastructure and Support, mentre presenta le nuove istanze Amazon EC2 Mac Instances durante l'Infrastructure Keynote del re:Invent 2020.

mac1.metal — uno sguardo alle specifiche

A questo punto — assumendo che non abbiate conoscenza delle specifiche hardware delle istanze Amazon EC2 Mac — potreste chiedervi quali siano i tagli supportati. Beh, per quanto riguarda l'immediato, scordatevi di poter scegliere: AWS per ora rende disponibile solo un taglio di istanza Mac. Le specifiche hardware ci dicono che le istanze mac1.metal sono alimentate da un processore Intel Coffee Lake a 3.2 GHz — potenziabile sino a 4.6 GHz — e 32 GiB di memoria. Come spiegato da Jeff Barr nel blog di AWS, le istanze girano in VPC, includono Elastic Network Adapter, e sono ottimizzate per la comunicazione con volumi EBS, infine supportano operazioni intensive di I/O.

Durante la mia routine giornaliera, il mio partner di lavoro è un laptop macOS che ho prontamente aggiornato al nuovo sistema operativo Apple: Big Sur. Di fatto ad oggi non ho notato particolari miglioramenti, tuttavia mantenere il sistema aggiornato rimane una best practice, almeno per quanto riguarda il computer di lavoro. In questo senso le Istanze AWS EC2 Mac ci sono fornite con una limitazione: supporto solo per OS Mojave e Catalina. Le AMI di questi sistemi ci vengono fornite con già preinstallati AWS CLI, tool da riga di comando per XCode, Homebrew e agent di SSM. AWS sta lavorando attivamente al supporto di Big Sur e anche ai Chip Apple M1.

Nel pratico, un caso d'uso

Ora, diamo uno sguardo a ciò che ci interessa di più: un caso d'uso pratico!

Ho cominciato la mia carriera lavorativa come sviluppatore, e immagino che diversi sviluppatori abbiano pensato, a loro volta, ad un'associazione tra questo annuncio e la possibilità di sviluppare, testare e firmare applicazioni macOS e iOS in cloud.

Durante l'ultimo anno, il mio team sta sviluppando un'applicazione Desktop Open-Source che gestisce credenziali in locale per permettere l'accesso ad Ambienti complessi in Cloud. Il nostro progetto è scritto in Typescript, interpretato da Node.js. Utilizziamo Angular come framework di sviluppo, che viene fatto girare on-top ad Electron per garantirci compatibilità multi-piattaforma

Electron viene fornito di un sistema di compilazione nativo, chiamato Electron-Builder, che permette di aggiungere script personalizzati di build al file package.json, file che contiene inoltre, come di consueto, le dipendenze di progetto. Abbiamo creato diversi script personalizzati per generare i binari Linux, Windows e macOS.

Per sviluppare il binario macOS, lo script deve avere accesso ad un Certificato di Firma e alla Password di Autenticazione (Notarisation) Apple. Questi permettono, rispettivamente, di firmare e autenticare il binario. Abitualmente siamo soliti salvare dati sensibili come i segreti nel nostro Keychain di macOS, lanciare gli script di build in locale e caricare manualmente gli artefatti sul nostro repository di GitHub, infine creare una nuova Release. Questo processo è, per sommi capi, una pratica comune adottata da molti sviluppatori per creare applicativi macOs e iOS

Questo processo è però lento, complicato e prono ad errori umani. Ma Hey! Sembra che una nuova opportunità si stagli davanti a noi! Quale caso d'uso migliore delle nuove istanze AWS EC2 Mac per

creare un binario macOS?

E' tempo di focalizzarci su come preparare il nostro banco di prova. Nelle prossime pagine andremo ad analizzare i seguenti passaggi:

- creazione di una nuova istanza mac1.metal pulita;
- accesso all'istanza;
- installazione dei pacchetti e dei tool necessari al processo di build;
- configurazione di una pipeline di build mediante il "buon vecchio zio" Jenkins.

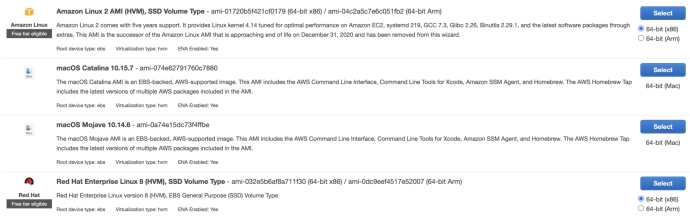
Lanciamo una istanza mac1.metal

La prima cosa che dobbiamo fare per configurare la nostra pipeline consiste nel creare l'istanza EC2 MAC su cui andremo a installare e configurare Jenkins.

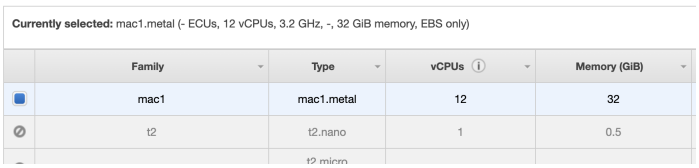
Buttiamoci sulla console AWS per cominciare!

Come per ogni altra istanza EC2, il wizard di lancio di mac1.metal parte dalla console di EC2, alla voce "Istanze". Come molti di voi sicuramente sanno, qui è possibile trovare una lista di istanze EC2 disponibili e — tra le altre cose — il menù necessario a lanciare questa nuova istanza.

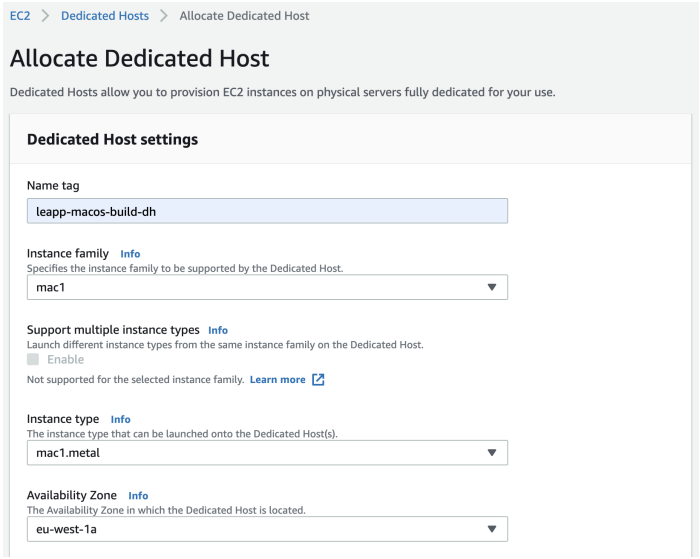
Dal 30 Novembre 2020, la lista di AMI disponibili si è arricchita infatti, ci mostra sia Mojave che Catalina, per i nostri scopi, scegliamo Catalina.



Nel momento in ci ci si trova a dover scegliere il tipo di istanza, solo una è disponibile al momento, chiamata mac1-metal. Come già sappiamo è caratterizzata da 12 vCPU a 3.2GHz e 32GiB di memoria. Se la vediamo come se fosse la nostra macchina di lavoro, non suona così strano alla fine!



Nel resto del wizard, possiamo trattare la nostra istanza Mac come ogni altra istanza; l'unica cosa su cui dovremmo soffermarci è la voce "tenancy". Dobbiamo lanciare l'istanza su una Macchina Dedicata che possiamo richiedere — in una tab separata — durante il processo di creazione dell'istanza. Anche per la Macchina Dedicata, dovremo specificare mac1.metal come tipo di istanza da poterci lanciare sopra.



Per questa proof-of-concept, ho deciso di lanciare l'istanza in una subnet pubblica e abilitare l'auto assegnazione dell'IP.

Per permettere l'installazione di un peso massimo come XCode — necessario per alcuni importantissimi strumenti di build — bisogna attaccare all'istanza un volume EBS di root con una dimensione appropriata; nel mio caso, ho scelto di agganciare un volume da 100GiB.

E per quanto riguarda i tipi di accesso?

Prima di dedicarci ai tipi di accesso delle istanze Mac, ricordiamoci di configurare i Security Group attivando le regole che permettono l'accesso alla macchina via SSH e VNC. Inoltre ho configurato una regola per l'accesso alla macchina via browser tramite porta 8080, per intenderci, quella legata al servizio di Jenkins. Siccome ho deciso di lanciare l'istanza in una subnet pubblica, ho ristretto l'accesso soltanto al mio IP corrente.

NOTA: funziona, ma per favore tenete conto che il setup da me proposto — in termini di rete e sicurezza — non può essere considerato production-ready! Per rendere il tutto più sicuro, è sufficiente spostare l'istanza Mac in una subnet privata, collegata ad un server openVPN in una subnet pubblica; in questo modo, l'istanza mac1.metal non è più esposta direttamente su internet, accedibile invece tramite il server OpenVPN.

Nell'ultimo passaggio del wizard di creazione dell'istanza, dobbiamo decidere se utilizzare una chiave .pem esistente o selezionarne una nuova per accedere alla macchina via SSH. Questo è il primo tipo di accesso disponibile e, non è per niente differente da un accesso SSH standard ad una istanza Linux; infatti, il nome utente per le macchine Mac è proprio ec2-user.

Se avete bisogno di un accesso con interfaccia grafica all'istanza — necessario quando andremo ad

installare i pacchetti software e i tool — potete utilizzare VNC; se siete utenti macOS, potete sfruttare VNC Viewer per configurare una connessione con l'istanza mac1.metal.

Ma aspettate — come ci mostra l' AWS Technical Evangelist Sébastien Stormacq in questo video — dobbiamo configurare una cosa molto importante prima di poter accedere all'istanza. In particolare, dobbiamo scegliere una password per l'utente ec2-user ed abilitare il server VNC. Questo è il Gist di Github che ho seguito e che si è dimostrato indispensabile per me:

```
# YouTube (english) : https://www.youtube.com/watch?v=FtU2_bBfSgM
# YouTube (french) : https://www.youtube.com/watch?v=VjnaVBnERDU

#
# On your laptop, connect to the Mac instance with SSH (similar to Linux instances)
#
ssh -i ec2-user@

#
# On the Mac
#

# Set a password for ec2-user

sudo passwd ec2-user

# Enable VNC Server (thanks arnib@amazon.com for the feedback and tests)

sudo /System/Library/CoreServices/RemoteManagement/ARDAgent.app/Contents/Resources/kickstart \
-activate -configure -access -on \
-configure -allowAccessFor -specifiedUsers \
-configure -users ec2-user \
-configure -restart -agent -privs -all

sudo /System/Library/CoreServices/RemoteManagement/ARDAgent.app/Contents/Resources/kickstart \
-configure -access -on -privs -all -users ec2-user

exit

#
# On your laptop
# Create a SSH tunnel to VNC and connect from a vnc client using user ec2-user and the password you defined.
#

ssh -L 5900:localhost:5900 -C -N -i ec2-user@
```

```
36
37  # open another terminal
38
39  open vnc://localhost
40
41  #
42  # On the mac, resize the APFS container to match EBS volume size
43  #
44
45  PDISK=$(diskutil list physical external | head -n1 | cut -d" " -f1)
46  APFSCONT=$(diskutil list physical external | grep "Apple_APFS" | tr -s " " | cut -d" " -f8)
47  sudo diskutil repairDisk $PDISK
48  # Accept the prompt with "y", then paste this command
49  sudo diskutil apfs resizeContainer $APFSCONT 0
50
51
52
53
54
55
```

NOTA: le linee 45-49 sono fondamentali nel senso che, senza di esse, non sareste in grado di installare Xcode; la dimensione iniziale del container APFS non è grande abbastanza per contenere Xcode.

E questo è tutto per quello che riguarda i tipi di accesso.

Installiamo pacchetti e tool

La pipeline di build che andremo a configurare necessita di alcuni step preliminari che dovremo soddisfare. Ok, ma quali sono?

- Un' installazione valida di Java
- L'applicazione Xcode, perchè i tool da linea di comando di default non forniscono altool, che è necessario per autenticare la app.
- Il servizio di Jenkins

Quando possibile, ho preferito utilizzare Brew per l'installazione dei pacchetti. In particolare, l'ho utilizzato per installare Java e Jenkins; quindi, ho provveduto ad installare Xcode dall' App Store, utilizzando le mie credenziali Apple.

Prima di tutto Xcode, il processo è semplice: apriamo App Store, cerchiamo Xcode e lo installiamo.

Ora è il turno di Java. Ho optato per Java AdoptOpenJDK versione 11, che può essere installato tramite Brew in questo modo:

```
# Update brew first of all
brew update

# Add adoptopenjdk/openjdk repository
brew tap adoptopenjdk/openjdk

# Install Java AdoptOpenJDK 11
brew install --cask adoptopenjdk11
```

Per quanto riguarda Jenkins, invece, ho utilizzato il seguente comando:

```
brew install jenkins-lts
```

Una volta installato, si può rendere Jenkins accessibile dall'esterno modificando il file /usr/local/opt/jenkins-lts/homebrew.mxcl.jenkins-lts.plist. In particolare, dobbiamo cambiare l'opzione –httpListenAddress da 127.0.0.1 a 0.0.0.0. Infine, lanciamo semplicemente il servizio con questo comando:

```
brew services start jenkins-lts
```

Configurazione iniziale di Jenkins

La configurazione iniziale di Jenkins prevede 3 passaggi importanti:

- sbloccare Jenkins — in questo step, dobbiamo recuperare la password scritta dentro il file /Users/ec2-user/.jenkins/secrets/initialAdminPassword;
- installare i plugin — in aggiunta a quelli pre-installati, attiviamo NodeJS, Git Parameter, e GitHub;
- creare un utente admin — infine, creato l' utente amministratore, inseriamo credenziali, nome completo e email.

Credenziali & Segreti

Prima di concentrarci sul processo di build, concentriamoci sul generare tutte le credenziali e i segreti necessari a scaricare il codice dal repository, firmare il binario, autenticarlo, e infine "pushare" l'artifact sul repository GitHub come nuova release draft.

Quindi, quali credenziali e segreti ci servono per questo processo di build?

- GitHub personal access token
- Apple signing certificate (in formato base64)
- Apple notarisation password

Personalmente li ho settati come segreti globali a partire da <jenkins-url>/credentials/store/system/. Da qui, possiamo aggiungere credenziali e segreti rispettando i seguenti tipi:

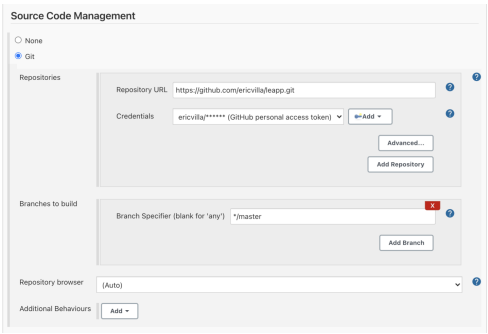
Name	Kind
ericvilla/****** (GitHub personal access token)	Username with password
Apple notarisation password	Secret text
signing-certificate-base64.p12	Secret text
GitHub personal access token 2	Secret text

Il segreto **GitHub personal access token 2** è differente dal primo, nel senso che viene utilizzato per inviare l'artifact verso il repository GitHub, mentre il primo per ottenere il codice dal repository.

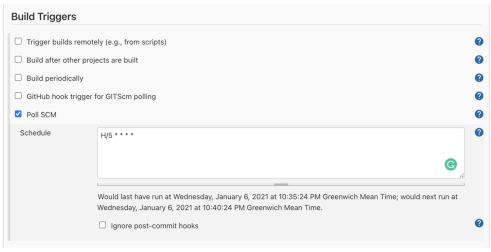
Configurazione del processo di Build

Possiamo lanciare il wizard di configurazione del processo di build premendo "New Item" dalla barra laterale a sinistra della Dashboard. Per quanto concerne il wizard, vedremo nel dettaglio le sezioni Source Code Management, Build Triggers, Build Environment Bindings, e Build.

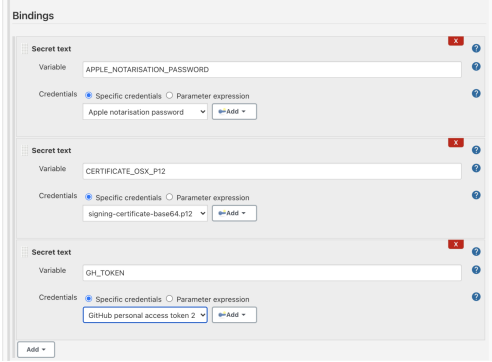
Nella sezione Source Code Management, dobbiamo specificare le coordinate del nostro repository GitHub, aggiungere le credenziali di accesso, l'URL del repository, e il ramo dal quale Jenkins recupererà il codice.



ho deciso di impostare il timer di Jenkins per i nuovi commit ogni 5 minuti, in modalità polling; l'alternativa consiste nel settare in Jenkins un webhook invocato direttamente da GitHub, in modalità push.



Prossimo passaggio: Variabili d'Ambiente, che ho descritto come **Build Environment Bindings**. Qui dobbiamo semplicemente assegnare le credenziali e i segreti precedentemente configurati alle variabili d'ambiente necessarie ai prossimi passaggi del processo di Build.



Finalmente, aggiungiamo i passaggi di build! Ho pensato di dividerli in 5 fasi, di fatto 5 “Execute shell commands”, che ho voluto illustrarvi qui sotto.

```
# Step 1
chmod +x ./scripts/add-osx-cert.sh

# Step 2
./scripts/add-osx-cert.sh

# Step 3
npm install

# Step 4
npm run rebuild-keytar

# Step 5
npm run dist-mac-prod
```

A parte gli step 3 e 4, potreste chiedervi cosa sono gli script add-osx-cert.sh e dist-mac-prod. Perciò, voglio mostrarvi la loro implementazione.

- project-root-folder/scripts/add-osx-cert.sh

```
#!/usr/bin/env sh

KEY_CHAIN=build.keychain
CERTIFICATE_P12=certificate.p12

echo "Recreate the certificate from the secure environment variable"
echo $CERTIFICATE_OSX_P12 | base64 --decode > $CERTIFICATE_P12

echo "security create-keychain"
security create-keychain -p jenkins $KEY_CHAIN
echo "security list-keychains"
security list-keychains -s login.keychain build.keychain
echo "security default-keychain"
security default-keychain -s $KEY_CHAIN
echo "security unlock-keychain"
security unlock-keychain -p jenkins $KEY_CHAIN
echo "security import"
security import $CERTIFICATE_P12 -k $KEY_CHAIN -P "" -T /usr/bin/codesign;
echo "security find-identity"
security find-identity -v
echo "security set-key-partition-list"
```

```
security set-key-partition-list -S apple-tool:,apple:,codesign:, -s -k jenkins $KEY_C
HAIN

# Remove certs
rm -fr *.p12
```

- dist-mac-prod — è uno script custom da aggiungere alla sezione script del file package.json

```
"dist-mac-prod": "rm -rf electron/dist && rm -rf release && rm -rf dist && tsc --p ele
ctron && ng build --aot --configuration production --base-href ./ && mkdir -p electro
n/dist/electron/assets/images && cp -a electron/assets/images/* electron/dist/electro
n/assets/images/ && electron-builder build --publish=onTag",
```

Sempre per rimanere focalizzati, la parte importante da sapere nello script dist-mac-prod è

```
electron-builder build --publish=onTag
```

che permette di lanciare la build del binario.

Dentro il file package.json, c'è un'altra parte molto importante che va specificata, ed è quella inclusa nella sezione "build"; eccolo mostrato qui di seguito.

```
# ...
"build": {
        "publish": [
            {
                "provider": "github",
                "owner": "ericvilla",
                "repo": "leapp",
                "releaseType": "draft"
            }
        ],
        "afterSign": "scripts/notarize.js",
# ...
```

Come potete vedere, sto richiedendo di inviare l'artifact al repository ericvilla/leapp — un fork di https://github.com/Noovolari/leapp — come nuova release in DRAFT.

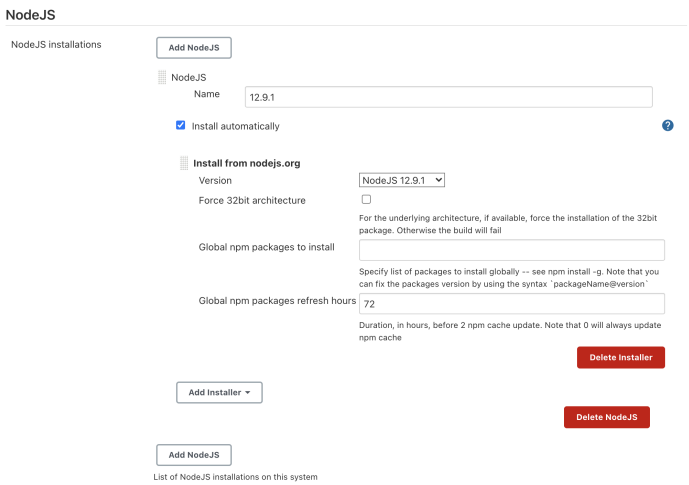
In più, ho specificato di lanciare lo script notarize.js subito dopo la firma. Questo script è responsabile per l'autenticazione dell'applicazione macOS, ed è implementato come segue:

```
const { notarize } = require('electron-notarize');

exports.default = async function notarizing(context) {
  const { electronPlatformName, appOutDir } = context;
  if (electronPlatformName !== 'darwin') {
    return;
  }

  const appName = context.packager.appInfo.productFilename;
```

```
  return await notarize({
    appBundleId: 'com.noovolari.leapp',
    appPath: `${appOutDir}/${appName}.app`,
    appleId: "mobile@besharp.it",
    appleIdPassword: process.env.APPLE_NOTARISATION_PASSWORD ? process.env.APPLE_NOTA
RISATION_PASSWORD :
      "@keychain:Leapp",
  });
};
```

Dipende dal pacchetto electron-notarize e cerca la variabile d'ambiente APPLE_NOTARISATION_PASSWORD; se non presente, la libreria assume che stiamo eseguendo la build sulla nostra macchina locale. Inoltre, ricerca la chiave appleIdPassword all'interno del Keychain di sistema.

Infine ma non meno importante, ho dovuto installare una versione di Node.js da dentro Jenkins per permettere alla soluzione di funzionare correttamente. Per fare questo, andiamo alla sezione "Manage Jenkins", accessibile dal menù laterale di sinistra. Una volta lì, premiamo "Global Tool Configuration"; quindi installiamo la versione di Node.js che ci serve. Nel mio caso, ho scelto la versione 12.9.1.



Ed eccoci qui, alla fine della configurazione del processo di Build!

Se il servizio di Jenkins è funzionante e il processo di Build è configurato correttamente, nuovi push sul repository GitHub scateneranno la Build.

Se la build avrà successo, potrete trovare la nuova release in Draft all'indirizzo https://github.com/<username>/<repository-name>/releases path.

Riassumiamo un pò

Nel leggere questo blog post tutto ci sarà sembrato chiaro e semplice (beh, almeno è quello che spero 🙂), ma durante i miei primi test sono incappato in diversi problemi diverse volte, e mi piacerebbe riassumerli qui.

Prima di tutto, lo sapevate che per poter lanciare un'istanza Mac dovete allocare un Dedicated Host per almeno 24h? Forse si. Ma, nel contesto di questo use case, è cruciale la possibilità di poter lanciare i processi di Build solo quando ci serve; potreste dirmi "ok, ma tu hai configurato Jenkins per cercare nuovi commit in modalità polling", e io vi risponderei che avete ragione. Infatti, normalmente ci basiamo su servizi container-based come AWS CodeBuild per lanciare le Build, diciamo, on-demand. Potrebbe sembrare più conveniente lanciare le Build localmente, o automaticamente configurando Jenkins come abbiamo fatto, ma sulla nostra macchina locale.

Perchè sto dicendo questo?

Semplice. Una istanza Mac con Dedicated Host non è affatto economica: costa $1.083 all'ora, il che significa costi fissi per almeno $26 ogniqualvolta decido di lanciarne una nuova (Dedicated Host allocato per almeno 24h). A causa del tempo minimo di allocazione, IMO non credo sia possibile — o almeno molto difficile — definire strategie di risparmio sui costi, come ad esempio avviare l'istanza di CI/CD solamente durante l'orario di lavoro, cioè. 8 ore al giorno.

E non è tutto.

Se decidiamo di lanciare una istanza Mac, dobbiamo considerare il fatto che spesso gli health check falliscono; questo significa — per le AMI con EBS — che è necessario fermare la macchina, aspettare fino a che il Dedicated Host esce dallo stato pending, infine, finalmente, rilanciare l'istanza. E tutto questo cosa significa? Un sacco di tempo perso.

Sto usando un tono forse un pò polemico, main maniera costruttiva. Penso che si tratti di un grosso annuncio, e che il servizio abbia in sé un grande potenziale. Si, potenziale, perchè ad oggi, mi ha dato l'impressione di essere ancora molto immaturo.

Al di là di queste considerazioni, il processo per configurare una pipeline con Jenkins è esattamente lo stesso rispetto ad una macchina locale, e i diversi modi di accesso all'istanza Mac ci hanno fornito un'esperienza completa.

Questo è tutto amici!

Ed eccoci qui.

I miei ringraziamenti vanno al tutto il mio team, in modo speciale ad Alessandro Gaggia, che mi ha dato supporto morale e tecnico durante la configurazione della Pipeline di Build.

Spero che questo blog post vi sia stato utile, e che vi abbia suscitato molte idee e quesiti. In quel caso, non esitate a contattarmi; sarà un piacere condividere considerazioni in merito a questo o altri use-case!

Ci si vede nei prossimi articoli!

Eric Villa

Senior DevOps Engineer @ beSharp

Get in touch

beSharp.it
proud2becloud@besharp.it