Installare plug-in custom su RDS con le Custom Resources di CloudFormation

29 Ottobre 2021 - 11 min. read

Amazon RDS | AWS CloudFormation | AWS Lambda | CloudFormation Custom resources | Database | Infrastructure as Code (IaC)

La **gestione e l'amministrazione dei database** sono sempre stati compiti molto delicati sia per server on premise, sia sul cloud. Lo sforzo richiesto per configurare e mantenere un database è molto alto: dalla configurazione della rete, alla restrizione dei permessi per i vari utenti, gruppi e ruoli. Backup e aggiornamenti, inoltre, richiedono anch'essi tempo. C'è anche da considerare il fatto che **le varie configurazioni di un database possono cambiare nel tempo** e richiedere quindi revisioni e modifiche, anche sostanziali.

Sfruttare i **database *managed*** diminuisce considerevolmente lo sforzo di gestione e amministrazione aggiungendo anche caratteristiche fondamentali come **scalabilità**, **alta disponibilità**, **fault-tolerance** e **sicurezza**.

Ma non basta: in molti casi può sopraggiungere per un amministratore di databases la necessità di installare uno o più **plug-in** (o estensioni) per incrementare le potenzialità del database o aggiungere funzionalità. Alcuni esempi relativi al mondo PostgreSQL, possono essere l'installazione di PostGis per query che includono informazioni geografiche, o PGCrypto per avere funzionalità crittografiche come la generazione del salt o l'hashing di stringhe. Queste sono operazioni manuali e ricorrenti che richiedono tempo e attenzione già per un singolo database. Oggi, però, anche le aziende più piccole necessitano di più database, a seconda del numero di servizi e/o applicazioni che offrono. Ripetere le stesse azioni per configurare correttamente molti databases è

un'operazione time-consuming e rischiosa: la **ripetizione manuale** delle stesse configurazioni ha un'alta probabilità di incappare in **errori**.

Per queste ragioni, la nuova sfida è trovare **un modo per schierare molteplici database con un dato set di configurazioni, in modo automatico**. In tutto questo, l'**Infrastructure as Code (IaC)** ci viene in aiuto: strumenti come Terraform o Pulumi, ad esempio, possono essere usati per automatizzare la creazione di svariate risorse in ambienti diversi, sia in locale, sia sui Cloud providers supportati, il tutto descrivendo l'infrastruttura desiderata tramite codice. Solitamente, i cloud providers offrono le loro soluzioni proprietarie per lo IaC. La soluzione di AWS è **AWS CloudFormation**.

Utilizzando i servizi cloud combinati con templates IaC, possiamo costruire una soluzione riutilizzabile che, con poche azioni manuali, può creare svariati databases, completamente gestiti, in modo automatico.

Rimane comunque la necessità dell'azione manuale per l'installazione di plugin specifici.

A questo problema non c'è una soluzione standard a seconda del motore: scegliendo, ad esempio, tra PostgreSQL o MySQL, ci si trova di fronte a differenti versioni, ognuna delle quali supporta un insieme differente di plugins, ognuno, a sua volta, con diverse versioni.

In pratica, serve **trovare un set di configurazioni compatibili** che lavori insieme. Una volta trovato questo set, potremo sfruttare lo IaC per automatizzare completamente la creazione dei databases.

In questo articolo, proponiamo una possibile soluzione a questo problema, automatizzando la creazione di database con conseguente installazione di plug-ins a bordo di essi. Per gli esempi contenuti in questo articolo useremo **AWS** come cloud provider e **RDS**, con **PostgreSQL**, come servizio database.

Prima di sporcarci le mani con il codice CloudFormation, ci sono alcune cose che dobbiamo tenere in considerazione. Diamo per assunto che tutte le configurazioni relative all'ambiente in cui il database verrà messo siano già state fatte, non essendo centrali in questo articolo. In particolare, diamo per scontato che VPC, sottoreti, tabelle di routing e security groups siano già pronti. **Verranno presi come parametri in input nel nostro codice per l'infrastruttura.**

Partendo dalla creazione del database, ci sono tre risorse principali che sono necessarie alla creazione dell'istanza: un **subnet group** per gestire il networking del database, un **parameter group** per definire alcuni parametri specifici della famiglia di database scelta e un **option group** per configurare alcune caratteristiche specifiche del dato engine del database.

```
DBSubnetGroup:
   Type: 'AWS::RDS::DBSubnetGroup'
   Properties:
     DBSubnetGroupDescription: !Sub "${DBName}-db-subnet-group"
     SubnetIds: [!Ref PrivateSubnetA, !Ref PrivateSubnetB, !Ref Priva
teSubnetC]
     Tags:
       - Key: Name
         Value: !Sub "${DBName}-db-subnet-group"
```

Come possiamo vedere dal codice, utilizzando il subnet group, abbiamo messo il nostro database nelle sottoreti private e abbiamo configurato l'uso di Postgres13 con l'option group. Nel mentre, tramite il parameter group, abbiamo configurato un singolo parametro, ai fini della spiegazione, impostando il massimo numero di connessioni al database, limitandolo a 30.

```
DBOptionGroup:
   Type: "AWS::RDS::OptionGroup"
   Properties:
     EngineName: "postgres"
     MajorEngineVersion: 13
     # OptionConfigurations: [] # no options needed for PostgreSQL
     OptionGroupDescription: !Sub "${DBName}-db-option-group"
     Tags:
       - Key: Name
         Value: !Sub "${DBName}-db-option-group"
```

Ora è il momento dell'istanza per il database. Dato che questo è un semplice esempio, possiamo rimanere su configurazioni di base, utilizzando un'istanza db.m5.large con 20 GBs per lo storage (gp2). Poi possiamo configurare altri parametri addizionali quali:

nome dell'istanza, nome dell'utente master e la relativa password, i parametri relativi alla cifratura dello storage, come la chiave KMS e le finestre di backup e manutenzione preferite.

Insieme all'istanza del DB, ci sono alcune risorse aggiuntive che possiamo utilizzare per migliorare la sicurezza del nostro database. Abbiamo definito una chiave KMS, insieme al relativo alias, per criptare lo storage e un segreto dentro al Secrets Manager per contenere le credenziali di admin per l'accesso al DB. Inoltre, come ulteriore layer di sicurezza, potremmo anche impostare un meccanismo che fa ruotare la password molto frequentemente, ad esempio ogni singolo giorno. Anche queste risorse possono essere create tramite CloudFormation. Non entreremo, però, nei dettagli di ciò dato che non è l'oggetto principale di questo articolo.

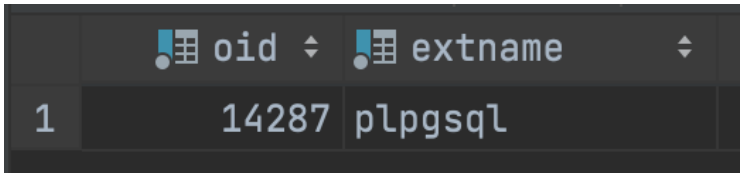
Giusto un'ultima aggiunta riguardo l'accesso al DB: in AWS c'è anche la possibilità, per i database che lo supportano, di utilizzare le credenziali IAM per accedere alle istanze di database. Ciò potrebbe risultare particolarmente utile per incrementare la sicurezza dell'infrastruttura dato che verrebbe ridotto drasticamente il numero di credenziali di accesso.

```
DBInstance:
   Type: 'AWS::RDS::DBInstance'
   Properties:
     DBInstanceIdentifier: !Ref DBName
     Engine: "postgres"
     DBInstanceClass: "db.m5.large"
     StorageType: "gp2"
     AllocatedStorage: 20
     DBParameterGroupName: !Ref DBParameterGroup
     OptionGroupName: !Ref DBOptionGroup
     DBSubnetGroupName: !Ref DBSubnetGroup
     VPCSecurityGroups: [!Ref DBSecurityGroup]
     MasterUsername: !Ref DBMasterUser
     MasterUserPassword: !Ref DBMasterUserPassword
     DBName: !Ref DBName
     Port: 5432
     AutoMinorVersionUpgrade: true
     CopyTagsToSnapshot: true
```

```
      Tags:
        - Key: Name
          Value: !Sub "${DBName}-db"
```

Ora che abbiamo lanciato il nostro template CloudFormation e il nostro database è stato creato, possiamo provare a connetterci e fare alcune queries. Per esempio, possiamo controllare il set di plug-ins installati con

```
SELECT * FROM pg_extension;
```



	oid	extname
1	14287	plpgsql

Un'altra cosa utile è controllare il set di estensioni permesse che può essere installato anche senza i permessi di amministratore. Possiamo controllarlo in due modi, il più semplice, dato che siamo già connessi al database, è la seguente query:

```
SHOW rds.extensions;
```

Altrimenti, se abbiamo la necessità di un plugin specifico, probabilmente è meglio verificare prima se è supportato o meno. Possiamo vedere la lista completa di plugin supportati a questo link.

PostgreSQL version 13 extensions supported on Amazon RDS

The following table shows PostgreSQL extensions for PostgreSQL version 13 that are currently supported on Amazon RDS. For more information on PostgreSQL extensions, see Packaging related objects into an extension.

Extension	13.4	13.3	13.2	13.1
address_standardizer	3.1.4	3.0.3	3.0.2	3.0.2
address_standardizer_data_us	3.1.4	3.0.3	3.0.2	3.0.2
amcheck	1.2	1.2	1.2	1.2
autoinc (contrib-spi)	1.0	N/A	N/A	N/A
aws_commons	1.1	1.1	1.1	1.1
aws_lambda	1.0	1.0	1.0	N/A
aws_s3.table_import_from_s3 aws_s3.query_export_to_s3	1.1	1.1	1.1	1.1
bloom	1.0	1.0	1.0	1.0
bool_plperl	1.0	1.0	1.0	1.0
btree_gin	1.3	1.3	1.3	1.3
btree_gist	1.5	1.5	1.5	1.5
citext	1.6	1.6	1.6	1.6
cube	1.4	1.4	1.4	1.4
dblink	1.2	1.2	1.2	1.2
dict_int	1.0	1.0	1.0	1.0
dict_xsyn	1.0	1.0	1.0	1.0

Come possiamo vedere, abbiamo la lista completa di plug-ins, insieme alla lista delle estensioni permesse che possiamo installare semplicemente con i normali permessi utente.

Dallo screenshot notiamo che abbiamo installato solo un plug-in nel nostro database, ma potremmo volerne aggiungere altri, come PostGis e PGCrypto. Per fare ciò potremmo lanciare semplicemente due queries:

```
CREATE EXTENSION IF NOT EXISTS postgis VERSION '3.0.3' CASCADE;
```

```
CREATE EXTENSION IF NOT EXISTS pgcrypto VERSION '1.3' CASCADE;
```

Come spiegato nell'introduzione, però, **questo approccio non è scalabile**. Un amministratore di databases può farlo per alcuni DB, ma al crescere del numero di plug-in da installare, il task richiederebbe troppo tempo e sforzo. In aggiunta, l'uomo è naturalmente più propenso a commettere errori. Un approccio manuale è, quindi, sempre sconsigliato per questo tipo di operazioni, sempre molto delicate.

Entriamo quindi nel vivo della nostra soluzione a questo problema: l'utilizzo delle **Custom Resource di CloudFormation**. Le Custom resources sono un modo per implementare logiche personalizzate di provisioning nel nostro codice IaC. Queste logiche vengono eseguite a ogni cambiamento dello stato di uno stack (creazione, aggiornamento, cancellazione). Nello specifico, queste risorse personalizzare eseguono i loro compiti tramite una funzione Lambda o un topic SNS. Per i nostri scopi, implementeremo la logica tramite una **Lambda**.

La Custom resource richiede semplicemente **un identificatore della funzione** che dovrà eseguire, ovvero il lambda ARN, e alcuni parametri aggiuntivi che verranno passati alla funzione per definire o modificare il suo comportamento. Si può decidere di avere più lambda a seconda del tipo di database che si sta creando, ad esempio, che sia un MySQL o un PostgreSQL, poiché il codice SQL effettivo e il set di plug-ins che dobbiamo installare possono essere diversi.

Detto questo, dobbiamo già avere la lambda creata, insieme al suo codice, poiché verrà eseguita non appena il database verrà creato. Anche la lambda potrebbe essere descritta nel nostro template IaC ma per semplicità la creeremo utilizzando la console AWS. Questo è utile perché siamo in grado di **testare e verificare il codice prima di utilizzarlo effettivamente nella custom resource**.

La lambda richiede alcune configurazioni: partendo dalle basi, la lambda stessa deve gestire un compito molto semplice quindi 512MB per la memoria saranno sufficienti.

Implementeremo il codice utilizzando Python3.8 come linguaggio e aggiungeremo un lambda layer per utilizzare la dipendenza psycopg2 per gestire la connessione con il database.

Ora ci sono ancora 3 parti fondamentali per finalizzare la lambda: permessi, rete e il codice vero e proprio. Partendo dal primo elemento della lista, dobbiamo solo creare il ruolo IAM per la lambda in grado di gestire il networking (in pratica, utilizzare le interfacce di rete) e leggere alcuni segreti all'interno del Secrets Manager.

Per connettersi al database**, la lambda deve essere eseguita nella stessa rete in cui è il database**, ovvero la VPC, e deve avere un modo per comunicare con esso. A tal fine, abbiamo bisogno di impostare **alcuni parametri aggiuntivi** nel nostro template CloudFormation. Dobbiamo posizionare la nostra custom resource in una sottorete pubblica, o meglio, con un NAT in modo tale che, non appena si avvierà, avrà un IP collegato che potrà essere utilizzato per comunicare con le altre risorse nella VPC. L'ultima cosa che dobbiamo configurare per quanto riguarda la rete sono i security groups: dobbiamo creare un security group per la lambda e poi consentire la comunicazione con il database utilizzando una regola in ingresso all'interno del nostro template CloudFormation per automatizzare questo processo per ogni database che verrà essere creato.

Finalmente, tutto ciò che riguarda la rete è impostato correttamente!

Ora la lambda può connettersi all'istanza del database. Prima di iniziare l'automazione della creazione del database insieme all'installazione dei plug-ins, iniziamo a scrivere il codice per il suo comportamento e verificare che tutto funzioni correttamente. Possiamo iniziare creando un metodo che elenchi le estensioni nel database, utilizzando la query di prima:

```
def get_extensions(cursor):
    extensions_query = "SELECT * FROM pg_extension"
    cursor.execute(extensions_query)
    return [row[1] for row in cursor.fetchall()]
```

Questo sarà utile per verificare effettivamente se abbiamo installato correttamente i plugin. Come possiamo vedere, nell'output c'è lo stesso plug-in che abbiamo visto in precedenza. Ora che siamo sicuri che la nostra custom resource è in grado di interagire con il database, possiamo creare il metodo che installerà effettivamente i plug-ins.

```
def create_extension(cursor, extension, version):
    extensions_query = 'CREATE EXTENSION IF NOT EXISTS "%s" VERSION "%
s" CASCADE;'
    cursor.execute(extensions_query, (AsIs(extension), AsIs(version
),))
```

Possiamo passare le estensioni, insieme alle loro versioni, alla lambda come mappa attraverso l'uso di un parametro nella definizione della custom resource nello IaC che vedremo in seguito.

L'ultima cosa che dobbiamo fare per finalizzare il nostro codice della lambda è inviare una risposta a un endpoint CloudFormation per informarlo della corretta esecuzione della custom resource. Ciò è necessario perché CloudFormation deve sapere quando termina la custom resource e, quindi, quando può continuare a creare le altre risorse nello IaC. Possiamo vedere l'implementazione in questa parte di codice:

```
def send_response(event: dict, context, response_status: str, respons
e_data: dict, no_echo=False, reason=None):
    logical_id = event.get('LogicalResourceId')
    digest = hashlib.sha256(logical_id.encode('utf-8')).hexdigest()
    physical_resource_id = logical_id + digest[:8]

    response_body = {
        'Status': response_status,
        'Reason': reason or f"See the details in CloudWatch Log Strea
m: {context.log_stream_name}",
        'PhysicalResourceId': physical_resource_id or context.log_stre
am_name,
        'StackId': event['StackId'],
        'RequestId': event['RequestId'],
        'LogicalResourceId': logical_id,
        'NoEcho': no_echo,
        'Data': response_data
    }
    try:
        response = http.request('PUT', event['ResponseURL'], body=json
```

```
.dumps(response_body))
    except Exception as e:
        logger.error(f"send_response failed executing http.request
(..): {e}")
```

Solo un commento su questa implementazione: per quanto riguarda i parametri di input per la funzione, abbiamo l'event e il context della funzione lambda, abbiamo poi lo stato della risposta che può essere "SUCCESS" o "FAILED". Poi possiamo avere i response data se dobbiamo inviare qualcosa a CloudFormation. Infine, abbiamo il parametro *no echo* in caso di informazioni sensibili e il parametro *reason* per commentare l'esecuzione della custom resource.

Ora possiamo finalizzare il nostro IaC con l'implementazione della custom resource. Qui il frammento di codice:

```
InstallPlugins:
    Type: Custom::ExecuteSQL
    Properties:
      ServiceToken: 'arn:aws:lambda:eu-west-2:364050767034:function:cu
stom-resource-demo'
      DBSecret: !Ref DBSecret
      PluginsMap:
        "PostGIS": '3.0.3'
        "pgcrypto": '1.3'
```

Si noti l'utilizzo del segreto DBSecret come parametro di input per la custom resource che contiene, in modo sicuro, tutte le informazioni necessarie per la connessione al database.

Ora che tutto è impostato correttamente, possiamo eseguire un test e creare il nostro database insieme alla custom resource che installerà i vari plug-ins. Iniziamo con un singolo database chiamato "demo1".

Non appena CloudFormation termina la creazione dello stack, abbiamo due modi per verificare l'installazione dei plug-ins. Possiamo provare a connetterci all'istanza ed eseguire la nostra query:

```
SELECT * FROM pg_extension;
```

oppure, se li abbiamo stampati nel codice della custom resource, possiamo semplicemente controllare i suoi log.

```
BEFORE:  ['plpgsql']
Installing plugin PostGIS version 3.0.3 ...
Installing plugin pgcrypto version 1.3 ...
AFTER:  ['plpgsql', 'postgis', 'pgcrypto']
```

Una volta verificato che tutto funzioni correttamente, possiamo iniziare a testare le altre caratteristiche di questa soluzione.

Innanzitutto, possiamo testare la sua capacità di scalare. Usiamo lo stesso IaC che abbiamo creato per creare due altri database: "demo2" e "demo3".

Come possiamo vedere dai logs della lambda, tutto funziona correttamente. Bene!

Ora possiamo testare un'altra caratteristica: la possibilità di recuperare il database. Possiamo creare uno snapshot utilizzando la console AWS e, non appena lo snapshot viene creato, possiamo provare a creare un database partendo da esso e verificare se abbiamo ancora i plug-ins. (Spoiler: sì, li abbiamo!)

Dopo che il database è stato ripristinato da un errore fittizio, abbiamo ancora i nostri dati e i nostri plug-ins!

Riassumendo

In questo articolo abbiamo visto come creare più database e installare su di essi alcune estensioni/plug-in, automatizzando il tutto tramite IaC, usando CloudFormation. Nel mentre, abbiamo scoperto la potenza e la facilità di utilizzo dei servizi Cloud per creare questo tipo di componenti infrastrutturali e, conseguentemente, per ottenere molte belle funzionalità quali alta disponibilità, scalabilità, backup automatici, fault-tolerance e sicurezza con uno sforzo minimo.

Per installare plug-ins/estensioni nel database, abbiamo esplorato il mondo delle custom resources per l'automazione delle azioni manuali tramite IaC. La speranza è di aver aperto uno spiraglio di luce su questo argomento che di solito viene visto come

oscuro e difficile. Nel frattempo, abbiamo spiegato come scrivere una lambda e come utilizzarla all'interno della rete per accedere ai vari databases.

In conclusione, abbiamo ottenuto la creazione automatizzata di una o più istanze di database nel cloud, con plug-ins pre-installati che possono essere piuttosto utili ai DBA.

Nella speranza che questo articolo sia stato abbastanza chiaro e uno spunto di ragionamento o discussione per qualcuno, se avete commenti o suggerimenti, non esitare a scriverci!

Matteo Goretti

DevOps Engineer @ beSharp. Appassionato di Cloud Computing e Intelligenza Artificiale, in particolare, Machine Learning e Deep Learning. Amo il trekking e la natura in generale. Mi rilasso con la mia chitarra, giocando ai videogames o guardando serie TV.
