Installing custom plugins on RDS using CloudFormation's Custom resource

29 October 2021 - 12 min. read

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

Database management and administration has always been a very delicate task both on-premises and on the cloud. The effort to configure and maintain a single database is pretty high: **network configurations**, restricting **permissions to various users, groups,** and roles, and the various **backups** and **updates** are tasks that take a lot of resources. Moreover, those configurations may **change over time**, causing the need to revise everything and modify somewhere, involving the risk attached to those modifications.

Having the DB as a managed service really lowers the effort required with features like scalability, high availability, fault tolerance, and security.

Given this, there are several cases in which a database administrator (DBA) may need to install **one or more plug-ins to enable some useful features** for the database. Some examples, specifically related to the PostgreSQL engine, may be the installation of **PostGis**, for location queries, and **PGCrypto** for cryptographic functions, like generating salts and hashing some strings. These are usually manual operations and they require a lot of time for a single database.

However, it's something that is not feasible in a real-world scenario in which, even small companies have multiple databases depending on the different services and applications that they have. Repeating the same steps to properly configure

everything for another database, or many more can be very **time-consuming** and requires a lot of effort. Moreover, manual configurations are always prone to **human errors**.

For these reasons, the next big challenge is to find a **reproducible set of configurations to deploy several other databases in an automated way**. This is where Infrastructure as Code (IaC) comes into play. Tools like Terraform or Pulumi can be used to automate the deployment of different resources in different environments, both local or on some supported cloud providers, describing the desired infrastructure with code. Usually, cloud providers offer their proprietary IaC solutions. The AWS one is CloudFormation.

Using cloud services combined with IaC templates, we are able to build a reusable solution that, with few manual actions, can automatically deploy several databases that are fully managed. This means: no more time spent patching the databases or replacing the old hardware! However, there is **still the need for manual actions** for the installation of specific plug-ins, therefore, involving all the issues related to the manual action on the configurations of a database.

For this problem, there is still no standard solution. Indeed, there is quite some variability. Starting from the selection of a given database engine, for example, PostgreSQL or MySQL, there are different versions of it, each one supporting different plug-ins and the different versions of them. In short, it is necessary to find a reproducible set of configurations that is compatible and work together. Once we have this reproducible solution we can take advantage of IaC to fully automate the deployment of several database instances.

In this article, we will propose a possible solution to this problem **automating the deployment of databases along with the installation of some plug-ins** over them. The examples in this article will use AWS, as a cloud provider, and PostgreSQL (RDS), as a database service.

Before getting our hands dirty with the Cloudformation code, there are some things that we need to take into account. We will assume that all the configurations relative to the environment in which the database will be installed are already done since they are out of the scope of this article. For this reason, things like VPC, subnets, routing tables, and security groups are meant as already prepared and, therefore, will be taken as input parameters in our infrastructure as code.

Starting from the deployment of a database, three major resources are needed to create a database instance: a subnet group to handle the networking of the database, a parameter group to define some parameters specific to the given database family, and an option group to configure some specific features of the given database engine.

```
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"
```

As we can see from the template, we have placed the database instance inside the private subnets (subnet group) and we've configured the option group to use Postgres13. Meanwhile, with the parameter group, we have configured a simple parameter, for the sake of explanation, that is the maximum number of connections for the database.

```
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"
```

Now, it's the time of the actual database instance. Since this is just a proof of concept, we can just use very basic configurations so we can use a db.m5.large instance with 20 GBs of storage (gp2). Then we can set other additional parameters like instance name, database name, master user along with the master password, the parameters for the

encryption of the storage, like the KMS key, the preferred backup, and maintenance windows.

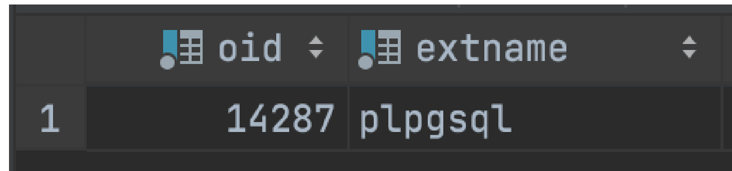
Along with the DB instance, there are just a few additional resources that can improve the security of our database. We have defined a KMS key, along with its alias, to encrypt the storage and a secret inside the Secrets Manager to store the admin credentials to access the DB. Moreover, we could also set up a rotation mechanism that rotates the password very frequently, like every single day. These kinds of resources could be *cloudformated* too, however, we won't get into the details of it since it's not the main topic of this article.

Just one last thing to add regarding DB access, inside AWS there is also the possibility of using IAM credentials to access DB instances. This feature may be useful to increase the security of the database since the number of credential pairs is drastically reduced.

```
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"
```

Now that we have deployed our CloudFormation template and our database is in place, we can try to connect to it and do some queries, for example, we can check the set of installed plug-ins with

```
SELECT * FROM pg_extension;
```



Another thing that may be very useful to check is the set of allowed extensions that can be installed even without admin permissions. We can check those in two ways. The easiest way, since we are already connected to the database, is to do this query:

```
SHOW rds.extensions;
```

However, if we need to have a specific plug-in, we may want to check beforehand if the plug-in is supported and, to do that, we can see the list of allowed extensions here.

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

As we can see, we have the full list of plug-ins, along with the list of the trusted extensions that we can easily install even without administrator permissions.

As we saw from the screenshot above, we have just one plug-in. We may want to add other extensions, like PostGis and PGCrypto. To do this we could just simply do a few queries to add them like:

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

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

However, as already explained, this is not a scalable approach. A DBA could do this for a few servers, but it would take too much time and effort to do more. Moreover, humans are error-prone so, a manual approach is discouraged for these kinds of operations that are usually very delicate.

Cloudormation's custom resources

The proposed solution is the use of **Cloudormation's custom resources**.

Custom resources are a way to implement custom provisioning logic in our IaC that will be executed as soon as there is a change in the state of a stack (creation, update, deletion). To be more specific, custom resources execute their work using either Lambda functions or SNS topics. For our purpose, we will implement the logic through a lambda function. The custom resource requires just an identifier of the function that will run, namely the lambda ARN, and some additional parameters that will be passed to the function to code its behavior.

Just a small addition, one can decide to have multiple lambdas depending on the type of database that is being created, like if it's a MySQL or a PostgreSQL, since the actual SQL code and the set of plug-ins that we need to install may be different.

Given this, we need to already have the lambda in place since it will run as soon as the database is created. Even the lambda could be coded in our IaC template but for simplicity, we will create it using the AWS console. This comes in handy because we are able to test and verify the code before actually using it in the custom resource.

The lambda requires some configurations: starting from the basics, the lambda itself needs to handle a very simple task so 512MB for the memory will be enough. We will implement the code using Python3.8 language and we will add a lambda layer to use psycopg2 dependency to handle the connection with the database.

Now there are still 3 core parts to finalize the lambda: permissions, networking, and the actual code. Starting from the first item of the list, we just need to create the IAM role for the lambda that can handle networking (basically, use network interfaces) and read some secrets inside the Secrets Manager.

To connect to the database, the lambda should run in the same network in which the database is placed, namely the VPC, and needs to have a way to communicate with it. For this purpose, we need to set some additional parameters in our CloudFormation template. We need to place our custom resource in a public, or better, natted subnet such that, as soon as it starts, it will have an attached IP that can be used to communicate with the other resources in the VPC. The last thing that we need to configure regarding the networking are security groups: we need to create a security group for the lambda and then allow the communication with the database using an ingress rule inside our CloudFormation template to automate this process for every database that will be created. Everything regarding the networking is now correctly set up!

Now that the function can connect to the database instance, we can start coding its behavior and test if everything is properly working before starting the automation of the DB creation along with the installation of the extensions. We can start by creating a method that lists the extensions in the database, using the query from above:

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

This will become in handy to actually check if we correctly installed the extensions. As we can see in the output there is the same plug-in that we saw earlier. Now that we are sure that our custom resource is capable of interacting with the database, we can create the method that actually installs the 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
),))
```

We can pass the extensions, along with their versions, to the lambda as a map through the use of a parameter in the custom resource definition in the IaC that we will see afterwards.

The last thing that we need to do to finalize our lambda code is to send a response to a CloudFormation endpoint to notify it about the correct execution of the custom resource. This is necessary because CloudFormation needs to know when the custom logic ends and, therefore, when it can continue to deploy the other resources in the IaC. We can see the implementation in this part of the code:

```
def send_response(event: dict, context, response_status: str, response_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 Stream: {context.log_stream_name}",
        'PhysicalResourceId': physical_resource_id or context.log_stream_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}")
```

Just a comment about this implementation. Regarding input parameters for the function, we have the event and the context of the lambda function, then we have the response status that can be either “SUCCESS” or “FAILED”. Then we can have response data if we need to send something back to CloudFormation. Finally, we have

the no echo parameter in case of sensitive information and the reason to comment about the execution of the custom resource.

Now we can finalize our IaC with the implementation of the custom resource. Here is the code snippet:

```
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'
```

Note the use of the secret DBSecret as an input parameter for the custom resource that contains, in a secure way, all the information needed for the database connection.

Now that everything is properly set up, we can run a test and deploy our database along with the custom resource that will install the plug-ins. Let's start with a single database called "demo1".
As soon as CloudFormation finishes deploying the stack we have two ways to check the installation of the plug-ins. Either we can try to connect to the instance and do our query:

```
SELECT * FROM pg_extension
```

or, if we print them in the custom resource code, we can simply check its logs.

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

Once we check everything is working properly, we can start testing the other features of this solution.

First, we can test its ability to scale. Let's use the same IaC that we have created to deploy two other databases: "demo2" and "demo3".

As we can see from the lambda logs, everything is properly working. Good!

Now we can test another feature: the ability to recover the database. We can create a snapshot using the AWS console, and, as soon as the snapshot is created, we can try to create a database from it and check if we still have the plug-ins. (Spoiler: yes, we do!)

After the database recovered from the failure we still have our data and our plug-ins!

To sum up

In this article we have seen how to create several databases and install some extensions/plug-ins on them, automating everything using a IaC tool like CloudFormation. In the process, we have discovered the power and easiness of using cloud services to deploy these kinds of infrastructural pieces and, along the way, achieving a lot of nice features like high availability, scalability, automated backups, fault tolerance and security with almost zero effort. To install plug-ins/extensions in the database, we explored the world of custom resources for automating manual actions using IaC. The hope is that we brighten up a bit on this topic that is usually seen as obscure and hard. In the meanwhile, we explained how to code a lambda and how to use it inside the network to access the various databases.

In conclusion, we obtained an automated deployment of one, or more, database instances in the cloud, with pre-installed plug-ins that may be pretty useful for DBAs.

Hoping this article was clear enough and has planted a seed for reasoning and discussion in someone's mind, if you have any comments or suggestions, feel free to write to us!

See you in 14 days here on **Proud2beCloud**!

Matteo Goretti

DevOps Engineer @ beSharp. Passionate about Artificial Intelligence, in particular, Machine Learning and Deep Learning, and interested in Cloud Computing. I love trekking and nature in general. I relax with my guitar, play video games, and watch TV series.

Copyright © 2011-2021 by beSharp srl - P.IVA IT02415160189