TROPOSPHERE: A BETTER WAY TO BUILD, MANAGE AND MAINTAIN A CLOUDFORMATION BASED INFRASTRUCTURE ON AWS

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

beSharp | 7 February 2020

Nowadays most of the modern SaaS applications are developed and deployed on Cloud providers and, in particular, Amazon Web Service, the first real Cloud provider, took and held the lead of this market due to the quality and the flexibility of its services.

AWS hosted Cloud Infrastructures keep getting larger and more complex with time in order to take full advantage of new services released by AWS. In fact, the number of services offered directly by Amazon is gargantuan and keeps growing every year. Using AWS services whenever possible instead of custom solutions deployed on EC2 virtual machines results in a huge decrease in the infrastructure setup and maintenance costs since Amazon is responsible for the deployment, Cloud optimization, security and maintenance of each service. Furthermore, most of the AWS services are designed to be highly available without any additional configuration, saving another significant configuration burden for the DevOps.

Using the AWS services as building blocks allows developer to create almost every type of application, for example, a typical serverless web application leverages Amazon Cognito for authentication, AWS Lambda/ApiGateway for the backend, DynamoDB for the database, SNS/SES for push and mail notifications to users, S3/Cloudfront for the frontend and SQS for internal queuing. However most applications are much more complicated than that (they often needs machine learning, datalakes, vpn connections to other services, different databases, batch processing and so on) and the number of different services and resources needed quickly escalates resulting in infrastructure so big and complicated that cannot be safely managed ‘by hand’ anymore. In fact, sometimes modifications to just one component (e.g. a security group or a routing table) to could result in unexpected side effects impacting several services and has the potential to take the whole Application offline.

In these cases, IaC (Infrastructure as Code) comes to the rescue. Through IaC it is possible to describe the whole AWS infrastructure writing regular code, so you can version it using Git just like any other code project. When the IaC code is executed it will create or update the infrastructure in order it to be exactly like you wrote in your code! If you need to change the infrastructure you update the IaC commit your change and rerun the code

If all this sounds too good to be true you are probably right! Every abstraction level we add to our software development flow comes with its own problems and IaC is no exception. The first problem we had when we decided to go with the IaC paradigm is the choice of the right tool, in fact there two main several IaC frameworks for AWS out there: Terraform and CloudFormation. We tried Terraform but found several issues which were a no-go for us:

- Terraform uses its own language which is also very limited: no loops and cycles are possible
- Sometimes Terraform fails to wait for resource creation resulting in difficult to debug errors
- It is possible for two developers to unknowingly run terraform at the same time resulting in infrastructure inconsistencies, if you want to use terraform a pipeline flow needs to be enforced for all projects at all times
- Rollbacks are often not carried out correctly.
- Changes often break at runtime because Terraform sometimes does not update resources in the right order.
- The resources are created using the AWS APIs and there is not a centralized place describing the actual state of the infrastructure
- Terraform run locally (or VM/container on AWS) so could be affected by network/hardware errors

CloudFormation, on the other hand, is a managed service by AWS: the user must simply write a YAML or JSON file describing all the infrastructure upload it on S3 or directly to Cloudformation and the service will take care of running it safely and statefully. Rollbacks are natively supported and it is also possible to execute “dry runs” of the template by creating a Change Set (analogously to terraform plan). In general, the execution of the template is much less error-prone than the one from terraform thanks to the service being AWS native. The only compelling Terraform use case is that of a multi-cloud infrastructure.

However, CloudFormation has its own drawbacks: YAML files are often very verbose and difficult to write and debug and like terraform do not support advanced logic and loops. Furthermore spitting a project in more files requires nested stacks that are difficult to integrate with Change Sets. So the next step is to generate the Cloudformation YAML templates using a more advanced language like python!

Here we have two alternatives AWS CDK and Troposphere. AWS CDK is new and extremely powerful and allows to declare complex infrastructure with very few lines of codes. However, being high level is also its biggest fault: some very low-level associations between resources are difficult to create and furthermore the output YAML template is difficult to read because all logical Ids of the resources are managed by CDK.

On the contrary, troposphere is really simple: it is just a Python DSL which maps CloudFormation Entities (all of them!) to Python classes and the other way round.

This gives us a very simple way to create a template that looks exactly like we want but is generated through a high level easily maintainable language. Furthermore, Python IDEs will help us fixing problems without even running the YAML template and the compilation step to YAML will break if we create inconsistent references.

To demonstrate the power of this workflow we show here how we can create a simple VPC with subnets, one for each Availability Zone.

First of all, let's look at the raw CloudFormation template:

```
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
```

We immediately notice that the code is readily readable and understandable even if it was automatically generated by a troposphere based script. As can immediately be seen most of the code is duplicated since we created 3 subnets with relative attachments to a routing table.

The python troposphere script which generated the script is the following:

```
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())
```

Running this script after installing Troposphere (pip install troposphere) will print the CF YAML shown above. As you can see the python code is much more compact and easy to understand. Furthermore, since Troposphere maps all the native cloudformation YAML functions (e.g. Ref, Join, GettAtt, etc.) we don't even need to learn anything new: every existing CF template can easily be converted in a Troposphere template.

Differently from plain CloudFormation with troposphere we can assign the various entities to python variables and use the python variables in the Ref and GettAtt functions in place of the logical CloudFormation names of the resource: in the example above we referenced the private subnet with Ref(private_subnet_route_table), not Ref('RouteTablePrivate'). This is a huge advantage because we don't need to remember the logical name while coding, the IDE will do that for us and warn us if the resource is not defined or has a different name.

Troposphere is also able to manage flawlessly nested stack and other complex multi Stack architecture through the Sceptre (https://github.com/Sceptre/sceptre) automation tool. However, instead of using Sceptre you can also write a custom deployment script, like we did in beSharp, to fully manage your deployment pipe and run automatic CloudFormation Drift changes check and evaluate the Change Set for all the nested templates before executing the template.

As a final remark troposphere is also able to manage the reverse flow: from a YAML template to python classes:

```
from cfn_tools import load_yaml
from troposphere import TemplateGenerator

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

This is very useful in situations where you need to dynamically update the infrastructure.

To conclude using Troposphere is a very simple way to reap all the advantages of CloudFormation together with the abstraction level provided by a modern programming language and it greatly simplifies CloudFormation code development and deployments. If you are interested in this topic do not hesitate to comment or reach us for further 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