CloudFormation Template Design

Raúl Estrada

Octubre 2020

Template design

CloudFormation templates are written in JSON and usually contain at least three sections (in any order): parameters, resources and outputs.

Unfortunately it is not possible to store these into separate files (with the exception of parameter values), so in this example we will work with a single template file named `main.json` .

Templates can be used locally or imported from a remote location (an S3 bucket is a common choice).

Parameters

Parameters add flexibility and portability to our Stack by letting us pass variables to it such as instance types, AMI ids, SSH keypair names and similar values which it is best not to hard-code.

Each parameter takes an arbitrary logical name (alphanumeric, unique within the template), description, type, and an optional default value. The available types are `String`, `Number`, `CommaDelimitedList`, and the more special AWS-specific type, such as `AWS::EC2::KeyPair::KeyName`, as seen in the preceding code.

The latter is useful for validation, as CloudFormation will check whether a key pair with the given name actually exists in your AWS account.

Parameters can also have properties such as `AllowedValues` , `Min/MaxLength` , `Min/MaxValue` , `NoEcho` and other (please see http://docs.aws.amazon.com/AWSCloudFormation/latest/UserGuide/parameters-section-structure.html).

There is a limit of 60 parameters per template.

Let us examine the parameters found at the top of our template:

Copy

```
"Parameters" : {
"vpcCidr" : {
"Description" : "VPC CIDR",
"Type" : "String"
},
"vpcName" : {
"Description" : "VPC name",
"Type" : "String"
},
"awsAvailabilityZones" : {
"Description" : "List of AZs",
"Type" : "CommaDelimitedList"
},
"publicCidr" : {
"Description" : "List of public subnet CIDRs",
"Type" : "CommaDelimitedList"
},...
"rdsInstanceClass" : {
"Description" : "RDS instance class",
"Type" : "String",
```

We have used the following:

- `CommaDelimitedList`, which we will conveniently query later with a special function
- `AllowedValues` and `MinValue` to enforce constraints
- `NoEcho` for passwords or other sensitive data
- Some AWS-specific types to have CloudFormation further validate input

You will notice that there are no values assigned to any of the preceding parameters.

To maintain a reusable template, we will store values in a separate file (`parameters.json`):

Copy

```
[
{
"ParameterKey": "vpcCidr",
"ParameterValue": "10.0.0.0/16"
},
{
"ParameterKey": "vpcName",
"ParameterValue": "CloudFormation"
},
{
"ParameterKey": "awsAvailabilityZones",
"ParameterValue": "us-east-1b,us-east-1c"
},
{
"ParameterKey": "publicCidr",
"ParameterValue": "10.0.1.0/24,10.0.3.0/24"
},
{
"ParameterKey": "privateCidr",
"ParameterValue": "10.0.2.0/24,10.0.4.0/24"
```

Copy

```
{
"ParameterKey": "privateCidr",
"ParameterValue": "10.0.2.0/24,10.0.4.0/24"
},
{
"ParameterKey": "rdsIdentifier",
"ParameterValue": "cloudformation"
},
{
"ParameterKey": "rdsStorageSize",
"ParameterValue": "5"
},
{
"ParameterKey": "rdsStorageType",
"ParameterValue": "gp2"
},
{
"ParameterKey": "rdsEngine",
"ParameterValue": "postgres"
},...
```

Resources

You are already familiar with the concept of resources and how they are used to describe different pieces of infrastructure.

Regardless of how resources appear in a template, CloudFormation will follow its internal logic to decide the order in which these get provisioned.

The syntax for declaring a resource is as follows:

Copy

```
"Logical ID" : {
"Type" : "",
"Properties" : {}
}
```

IDs need to be alphanumeric and unique within the template.

The list of CloudFormation resource types and their properties can be found here:
http://docs.aws.amazon.com/AWSCloudFormation/latest/UserGuide/aws-template-resource-type-ref.html

The max number of resources a template can have is 200. Reaching that limit, you will need to split a template into smaller ones and possibly look into nested stacks.

Back to our example, as per tradition we start by creating a VPC and its supporting elements such as subnets, Internet gateway and NAT gateway:

Copy

```
"Resources" : {
"vpc" : {
"Type" : "AWS::EC2::VPC",
"Properties" : {
"CidrBlock" : { "Ref" : "vpcCidr" },
"EnableDnsSupport" : "true",
"EnableDnsHostnames" : "true",
"Tags" : [ { "Key" : "Name", "Value" : { "Ref" : "vpcName" } } ]
}
},
"publicSubnet1" : {
"Type" : "AWS::EC2::Subnet",
"Properties" : {
"AvailabilityZone" : { "Fn::Select" : [ "0", {"Ref" : "awsAvailabilityZones"} ] },
"CidrBlock" : { "Fn::Select" : [ "0", {"Ref" : "publicCidr"} ] },
"MapPublicIpOnLaunch" : "true",
"Tags" : [ { "Key" : "Name", "Value" : "Public" } ],
"VpcId" : { "Ref" : "vpc" }
}
},
```

Copy

```
"Type" : "AWS::EC2::VPCGatewayAttachment",
"Properties" : {
"InternetGatewayId" : { "Ref" : "internetGateway" },
"VpcId" : { "Ref" : "vpc" }
}
},
"natEip" : {
"Type" : "AWS::EC2::EIP",
"Properties" : {
"Domain" : "vpc"
}
},
"natGateway" : {
"Type" : "AWS::EC2::NatGateway",
"Properties" : {
"AllocationId" : { "Fn::GetAtt" : ["natEip", "AllocationId"]},
"SubnetId" : { "Ref" : "publicSubnet1" }
},
"DependsOn" : "internetGatewayAttachment"
},
```

Note some of the `CloudFormation` functions used in the preceding code:

- `"Fn::Select"` in `"CidrBlock" : { "Fn::Select" : [ "0", {"Ref" : "publicCidr"} ] }`, which allows us to query the `CommaDelimitedList` type parameters we set earlier
- `"Fn::Join"`, for concatenating strings
- `"Fn::GetAtt"`, for retrieving resource attributes

Also, the `DependsOn` property of the `natGateway` resource allows us to set explicit conditions on the order of execution. In this case, we are saying that the Internet Gateway resource needs to be ready (attached to the VPC) before the NAT Gateway is provisioned.

After the VPC, let's add RDS:

Copy

```
    "rdsInstance" : {
    "Type" : "AWS::RDS::DBInstance",
    "Properties" : {
    "DBInstanceIdentifier" : { "Ref" : "rdsIdentifier" },
    "DBInstanceClass" : { "Ref" : "rdsInstanceClass" },
    "DBSubnetGroupName" : { "Ref" : "rdsSubnetGroup" },
    "Engine" : { "Ref" : "rdsEngine" },
    "EngineVersion" : { "Ref" : "rdsEngineVersion" },
    "MasterUserPassword" : { "Ref" : "rdsPassword" },
    "MasterUsername" : { "Ref" : "rdsUsername" },
    "StorageType" : { "Ref" : "rdsStorageType" },
    "AllocatedStorage" : { "Ref" : "rdsStorageSize" },
    "VPCSecurityGroups" : [ { "Ref" : "rdsSecurityGroup" } ],
    "Tags" : [ { "Key" : "Name", "Value" : { "Ref" : "rdsIdentifier" } } ]
    }}
```

Then add the ELB:

Copy

```
...
"elbInstance" : {
"Type" : "AWS::ElasticLoadBalancing::LoadBalancer",
"Properties" : {
"LoadBalancerName" : "cloudformation-elb",
"Listeners" : [ { "InstancePort" : "80", "InstanceProtocol" : "HTTP", "LoadBalancerPort" : "80", "Protocol"
"SecurityGroups" : [ { "Ref" : "elbSecurityGroup" } ],
"Subnets" : [ { "Ref" : "publicSubnet1" }, { "Ref" : "publicSubnet2" } ],
"Tags" : [ { "Key" : "Name", "Value" : "cloudformation-elb" } ]
}
}
```

And add the EC2 resources:

Copy

```
...
"launchConfiguration" : {
"Type" : "AWS::AutoScaling::LaunchConfiguration",
"Properties" : {
"ImageId" : { "Ref": "autoscalingGroupImageId" },
"InstanceType" : { "Ref" : "autoscalingGroupInstanceType" },
"KeyName" : { "Ref" : "autoscalingGroupKeyname" },
"SecurityGroups" : [ { "Ref" : "ec2SecurityGroup" } ]
```

We still use a `UserData` shell script to install the NGINX package; however, the presentation is slightly different this time. `CloudFormation` is going to concatenate the lines using a new line character as a delimiter then encode the result in `Base64` :

Copy

```
    "UserData" : {
    "Fn::Base64" : {
    "Fn::Join" : [
    "\n",
    [
    "#!/bin/bash",
    "set -euf -o pipefail",
    "exec 1> >(logger -s -t $(basename $0)) 2>&1",
    "yum -y install nginx; chkconfig nginx on; service nginx start"
    ]
    ]
    }
    }
    }
    }
```

We use `DependsOn` to ensure the RDS instance goes in before `autoScalingGroup` :

Copy

```
"autoScalingGroup" : {
"Type" : "AWS::AutoScaling::AutoScalingGroup",
"Properties" : {
"LaunchConfigurationName" : { "Ref" : "launchConfiguration" },
"DesiredCapacity" : "1",
"MinSize" : "1",
"MaxSize" : "1",
"LoadBalancerNames" : [ { "Ref" : "elbInstance" } ],
"VPCZoneIdentifier" : [ { "Ref" : "privateSubnet1" }, { "Ref" : "privateSubnet2" } ],
"Tags" : [ { "Key" : "Name", "Value" : "cloudformation-asg", "PropagateAtLaunch" : "true" } ]
},
"DependsOn" : "rdsInstance"
}
```

Outputs

Again, we will use these to highlight some resource attributes following a successful deployment. Another important feature of `Outputs` , however, is that they can be used as input parameters for other templates (stacks). This becomes very useful with nested stacks.

We add the `VPC ID` , `NAT IP` address and `ELB DNS` name as `Outputs` :

Copy

```
"Outputs" : {
"vpcId" : {
"Description" : "VPC ID",
"Value" : { "Ref" : "vpc" }
},
"natEip" : {
"Description" : "NAT IP address",
"Value" : { "Ref" : "natEip" }
},
"elbDns" : {
"Description" : "ELB DNS",
"Value" : { "Fn::GetAtt" : [ "elbInstance", "DNSName" ] }
}
}
```

Currently, a template can have no more than 60 Outputs.