The Red Head's Tech Blog: CloudFormation Mysteries Revealed

Posted by FALYCIA HANKINSON Nov 22, 2016

There are many methods of launching your infrastructure in AWS but fortunately for you, today is CFT day. CFT, as in CloudFormation Templates. As in, a document that lays out an entire blueprint of your virtual datacenter, a lonely S3 bucket, or a single instance.

Okay, so get your flashlight and let's go digging in the archives of docs.amazonaws.com... What? You don't want to do that?

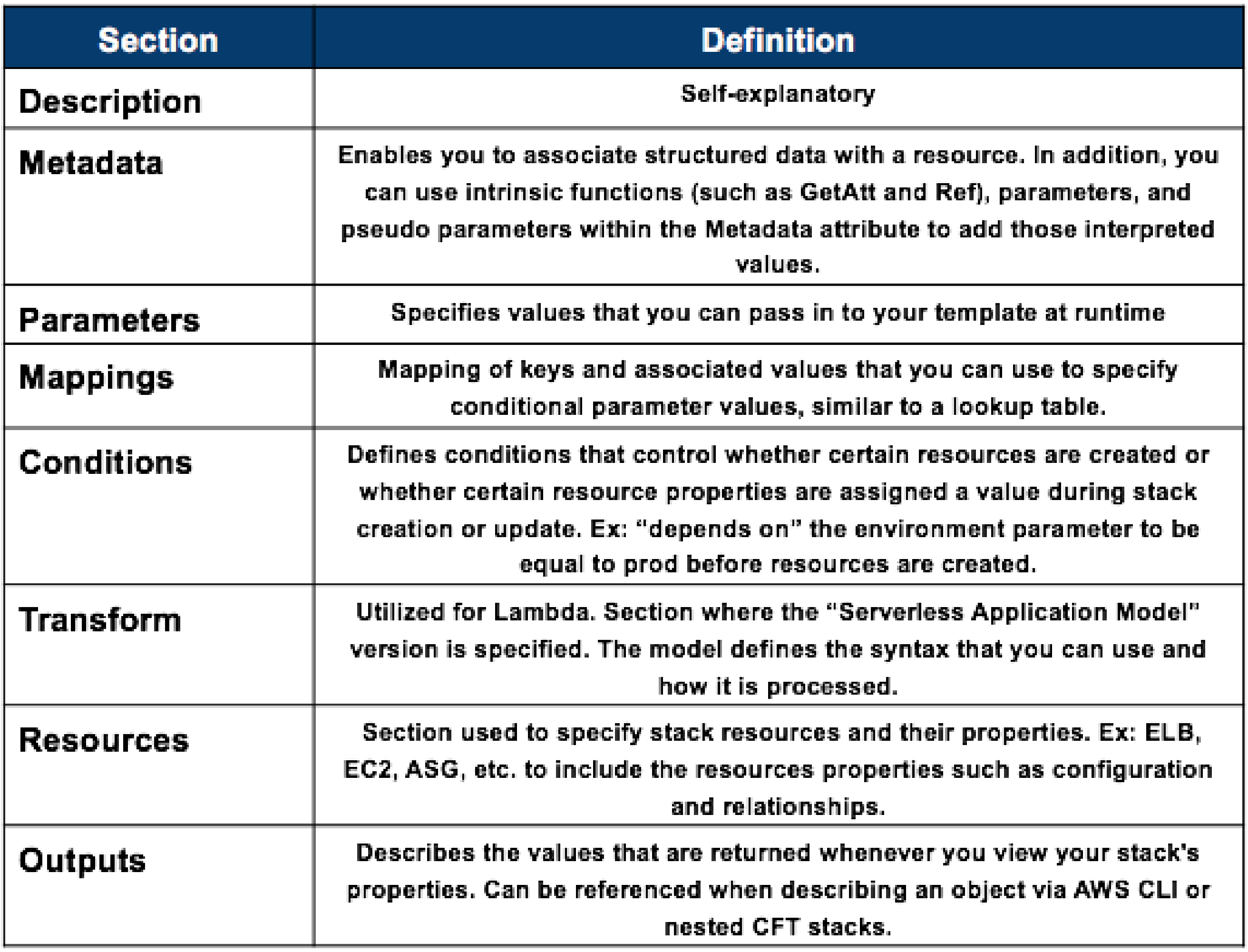
Lucky you!

Here is a more generalized overview of what a CFT is really made of:

Cloud Formation scripts are blueprints for your infrastructure and configuration set up. It is a full playbook of your environment. A two-dimensional layout of everything that you want to build. The structure uses a trickle down layout with key value pairs.

Pic Credit

A CFT Template is made up of sections. All sections EXCEPT resources are optional and do not have to be included.



Section	Definition
Description	Self-explanatory
Metadata	Enables you to associate structured data with a resource. In addition, you can use intrinsic functions (such as GetAtt and Ref), parameters, and pseudo parameters within the Metadata attribute to add those interpreted values.
Parameters	Specifies values that you can pass in to your template at runtime
Mappings	Mapping of keys and associated values that you can use to specify conditional parameter values, similar to a lookup table.
Conditions	Defines conditions that control whether certain resources are created or whether certain resource properties are assigned a value during stack creation or update. Ex: "depends on" the environment parameter to be equal to prod before resources are created.
Transform	Utilized for Lambda. Section where the "Serverless Application Model" version is specified. The model defines the syntax that you can use and how it is processed.
Resources	Section used to specify stack resources and their properties. Ex: ELB, EC2, ASG, etc. to include the resources properties such as configuration and relationships.
Outputs	Describes the values that are returned whenever you view your stack's properties. Can be referenced when describing an object via AWS CLI or nested CFT stacks.

Metadata (CFN INIT)

Use the AWS::CloudFormation::Init type to include metadata on an Amazon EC2 instance for the cfn-init helper script. If your template calls the cfn-init script, the script looks for resource metadata rooted in the AWS::CloudFormation::Init metadata key. All metadata types supported in Linux. Windows types are supported with conditions.

If you want to create more than one config key and to have cfn-init process them in a specific order, create a configset that contains the config keys in the desired order.

You can create multiple configsets, and call a series of them using your cfn-init script. Each configset can contain a list of config keys or references to other configsets. For example, the following template snippet creates three configsets. The first configset, test1, contains one config key named 1. The second configset, test2, contains a reference to the test1 configset and one config key named 2. The third configset, default, contains a reference to the configset test2.

```
JSON Example:
"AWS::CloudFormation::Init" : {
"configSets" : {
"ascending" : [ "config1" , "config2" ],
"descending" : [ "config2" , "config1" ]
    },
"config1" : {
"commands" : {
"test" : {
"command" : "echo \"$CFNTEST\" > test.txt",
"env" : { "CFNTEST" : "I come from config1." },
"cwd" : "~"
            }
        }
    },
"config2" : {
"commands" : {
"test" : {
"command" : "echo \"$CFNTEST\" > test.txt",
"env" : { "CFNTEST" : "I come from config2" },
                "cwd" : "~"
            }
        }
    }
}
```

Under "AWS::CloudFormation::Init type, there are specific sections that can be used to orchestrate your template setup:

- Commands section is required (String or array). All other are optional. Commands are processed in alphabetical order by name.
- Files section can be used to create files on an instance during setup. Can be done in-line or via URL.
- Groups and Users section can be used to create groups or users on the instance (Linux Only).
- Packages key is used to download and install packages onto the instance. Windows supports MSI installer only.
- Services allows you to specify services you want to enable/disabled or require restart.
- Sources key allows you to download an archive and unpack it to a directory.

```
JSON Command Example:
"commands" : {
    "test" : {
        "command" : "echo \"$MAGIC\" > test.txt",
        "env" : { "MAGIC" : "I come from the environment!" },
        "cwd" : "~",
        "test" : "test ! -e ~/test.txt",
        "ignoreErrors" : "false"
    }
}

YAML Command Example:
commands:
  test:
    command: "echo \"$MAGIC\" > test.txt"
    env:
      MAGIC: "I come from the environment!"
    cwd: "~"
    test: "test ! -e ~/test.txt"
    ignoreErrors: "false"
```

The above example snippets call the echo command IF the ~/test.txt file doesn't exist.

Parameters

Parameters allow you to specify input values that are defined at the beginning of the cloud formation creation process. In addition, they can be referenced using AWS CLI for stack updates for AMI refresh automation as well. But I do want to point out, you want to use these sparingly. No one wants to input 50 values when launching a cloudformation script. No one. So I recommend using parameters for values that are modified regularly (like AMI ID) or are used as mapping lookup values (line environment).

```
JSON:
"Parameters": {
   "Environment": {
      "AllowedValues": ["dev", "prod"],
      "Default": "dev",
      "Description": "What environment type is it (dev or prod)?",
      "Type": "String"
   },
   "ImageId" : {
      "Description" : "AMI-ID",
      "Type": "AWS::EC2::Image::Id”
   }
},

YAML:
Parameters:
  Environment:
    AllowedValues: - dev - prod
    Default: dev
    Description: What environment type is it (dev or prod)?
    Type: String
  ImageId:
    Type: AWS::EC2::Image::Id
    Description: AMI-ID
```

For a full list of parameter types go here.

Mapping

Mapping allows you to provide a structured key/pair section that you can look up values based off the structure itself.

Here is an example using the environment AND the region as a reference key for values:

```
JSON Example:
"Mappings": {
   "dev" : {
      "us-east-1" : {
         "Subnets" : "subnet-45e2ff6e, subnet-bb9a52cd, subnet-e029cab8",
         "AvailabilityZones": "us-east-1b,us-east-1c,us-east-1d",
         "ELBName" : "ISRMELB-Dev",
         "InstanceType" : "c4.4xlarge",
         "InstanceRole" : " ISRM-MyAwesomeRole-Dev",
         "S3Bucket" : “dev-bucket"
       },
      "us-west-2" : {
         "Subnets" : "subnet-45e2ff6e, subnet-bb9a52cd, subnet-e029cab8",
         "AvailabilityZones": "us-west-2b,us-west-2c,us-west-2d",
         "ELBName" : "ISRMELB-Dev-West",
         "InstanceType" : "c4.4xlarge",
         "InstanceRole" : " ISRM-MyAwesomeRole-Dev",
         "S3Bucket" : “dev-bucket-west"
       }
     },
   "prod" : {
      "us-east-1" : {
         "Subnets" : "subnet-42cada1b, subnet-e9484ac2, subnet-1b9a636d",
         "AvailabilityZones" : "us-east-1c, us-east-1a, us-east-1b",
         "ELBName" : "ISRMELB-Prod",
         "InstanceType" : "c4.4xlarge",
         "InstanceRole" : "ISRM-MyAwesomeRole-Prod",
         "S3Bucket" : "prod-bucket"
       }
     }
  },
```

To reference a value in the mapping you would use:

{ "Fn::FindInMap" : [{"Ref":"Environment"} ,{"Ref" : "AWS::Region"} ,"Subnets"] }

Here is another example using the built-in params structure:

```
JSON Example:
"Mappings": {
    "params": {
        "AvailabilityZones": {
            "dev": ["us-east-1d", "us-east-1a", "us-east-1b"],
            "prod": ["us-east-1a", "us-east-1b", "us-east-1d"],
            "prodwest": ["us-west-2a", "us-west-2b", "us-west-2c"]
          },
        "ImageId": {
            "dev": "ami-10e18307",
            "prod": "ami-e84d8085",
            "prodwest": "ami-ce20cbae"
          },
          "Subnets": {
            "dev": ["subnet-8caf9dd5", "subnet-24a1b953", "subnet-6eae8d45" ],
            "prod": ["subnet-3cbd9e17", "subnet-3d9fad64", "subnet-805348f7" ],
            "prodwest": ["subnet-a1037cd6", "subnet-a5c455fc", "subnet-0a6e376f" ]
          },
          "S3Bucket": {
            "dev": "isrm-test-dev",
            "prod": "isrm-test",
            "prodwest": "isrm-test-west"
          }
    }
},
```

To reference a value in the mapping structure above, you would use:

{"Fn::FindInMap": ["params","Subnets", { "Ref": "Environment" }] }

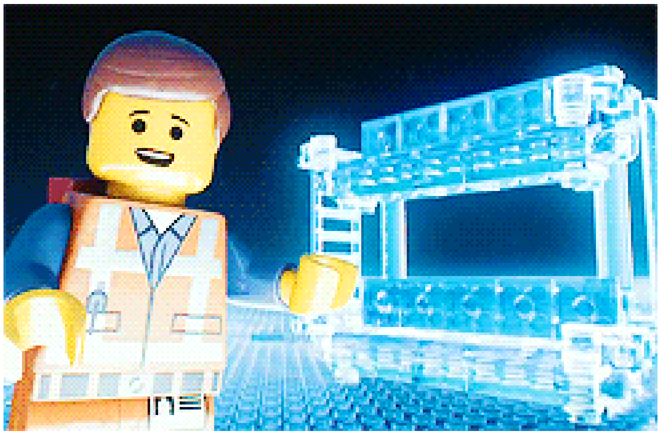
The biggest benefit you get from using the first structure is you do not have to create environments like "devwest" and "prodeast", you can use the same environment names and utilize the region where the cloudformation script is being launched.

Transform

This section isn't widely known or used since it is slightly new. The transform section is used for serverless-application setup (Lambda, API Gateway, DynamoDB, etc.). The transform section is where you would specify the version of the AWS Serverless Application Model (AWS SAM). This informs CloudFormation that this template contains AWS SAM resources that need to be 'transformed' to full-blown CloudFormation resources when the stack is created. The new AWS Serverless Application Model (AWS SAM) allows you to describe all of these components using a simplified syntax that is natively supported by AWS CloudFormation. You can find more info here.

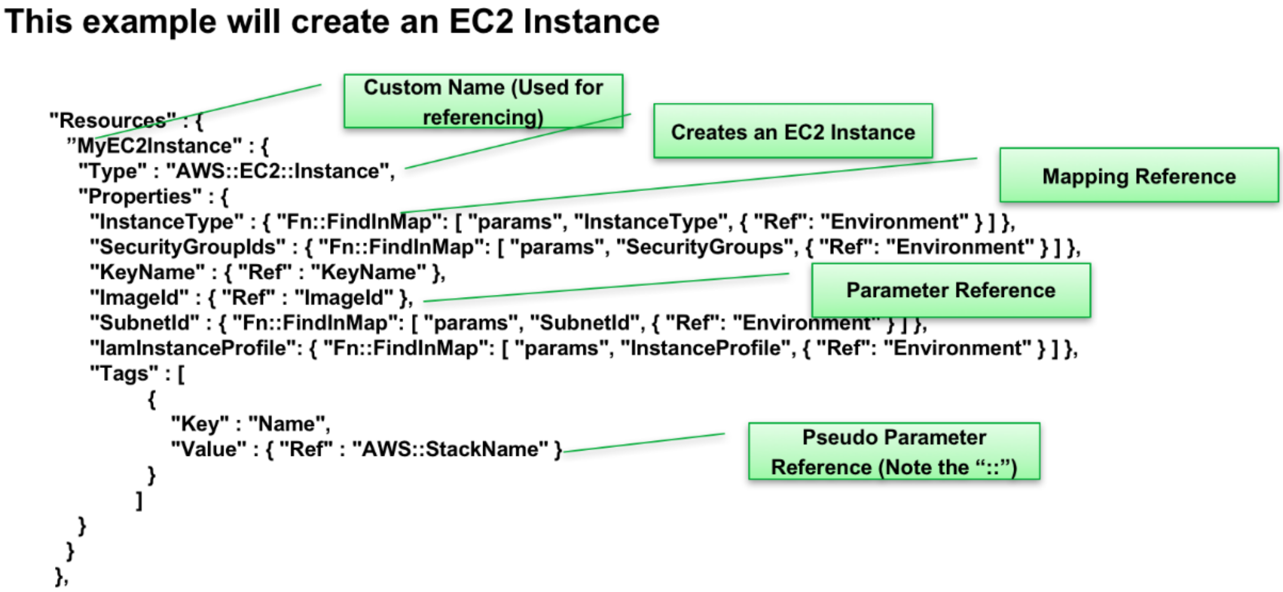
```
YAML Example:
AWSTemplateFormatVersion: '2010-09-09'
Transform: 'AWS::Serverless-2016-10-31'
Description: A starter AWS Lambda function.
Resources:
  ShowEnv:
    Type: 'AWS::Serverless::Function'
    Properties:
      Handler: lambda_function.lambda_handler
      Runtime: python2.7
      CodeUri: .
      Description: A starter AWS Lambda function.
      MemorySize: 128
      Timeout: 3
      Role: 'arn:aws:iam::99999999999:role/LambdaGeneralRole'
```

Well here we are at the "Big Kahuna", the "GodFather", the "Muckamuck" (yes, that is a word). This is where you create life! You get to build the instances, services, or infrastructure pieces that will rule the world!



Master Builder, get it? Well hopefully you get what I mean.

Resources are made up of a logical name (used for referencing in different CFT sections and logging in the CFT events).



This example will create an EC2 Instance

```
"Resources" : {
 ”MyEC2Instance" : {
  "Type" : "AWS::EC2::Instance",
  "Properties" : {
   "InstanceType" : { "Fn::FindInMap": [ "params", "InstanceType", { "Ref": "Environment" } ] },
   "SecurityGroupIds" : { "Fn::FindInMap": [ "params", "SecurityGroups", { "Ref": "Environment" } ] },
   "KeyName" : { "Ref" : "KeyName" },
   "ImageId" : { "Ref" : "ImageId" },
   "SubnetId" : { "Fn::FindInMap": [ "params", "SubnetId", { "Ref": "Environment" } ] },
   "IamInstanceProfile": { "Fn::FindInMap": [ "params", "InstanceProfile", { "Ref": "Environment" } ] },
   "Tags" : [
        {
          "Key" : "Name",
          "Value" : { "Ref" : "AWS::StackName" }
        }
      ]
  }
 }
},
```

Custom Name (Used for referencing)
Creates an EC2 Instance
Mapping Reference
Parameter Reference
Pseudo Parameter Reference (Note the “::”)

Avoid Hard Coded Values!

I don't know if you caught that bold red sentence up above but don't put hard coded values in a cloudformation resources section. It makes everyone sad. Your mother will be very disappointed in you. Use mappings, parameters, or ANY other means of inputting values into your resources. Just. Don't. Hardcode. This is the neutral zone, let's keep it that way.

Pseudo parameter? What is that up there? Well hold your horses, and I'll tell you. Pseudo parameters are nifty parameters you can reference that come with CFT out-of-the-gate. You do not have to assign your own values to these parameters because they are built-in with no extra charge!

From the example above you can see I used "AWS::StackName".

```
"Tags" : [
    {
        "Key" : "Name",
        "Value" : { "Ref" : "AWS::StackName" }
    }
]
```

This reference will return the string value that you, yes YOU, input into the console (or aws cli) when you created the stack.
Here are the currently allowed pseudo parameters:

AWS::AccountId
AWS::NotificationARNs

AWS::NoValue
AWS::Region
AWS::StackId
AWS::StackName

Note: "NoValue" removes the corresponding resource property when specified as a return value in the Fn::If intrinsic function. So for example, you can use the AWS::NoValue parameter when you want to use a snapshot for an Amazon RDS DB instance only if a snapshot ID is provided.

Conditions

You may have heard some word on the street that there are things called "conditions" when building a CFT stack. You would be correct. This is where that awesome word "intrinsic" comes into play again. Utilizing the intrinsic functions i.e. And, Equal, If, Not, Or, conditions allow you to "conditionally" create a resource. So for example let's say I only want to create a specific resource IF I am building a stack for the Prod environment, but if I am building a Dev stack, build a different resource. How the heck would I do that?

Easy-peasy. Build conditions for each scenario:

```
"Conditions" : {
  "CreateProdResources" : {"Fn::Equals" : [{"Ref" : "Environment"}, "prod"]},
  "CreateDevResources" : {"Fn::Equals" : [{"Ref" : "Environment"}, "dev"]}
},
```

Then I reference the condition in the resource that I want to launch for that specific environment:

```
"Resources" : {
    "EC2Instance" : {
      "Type" : "AWS::EC2::Instance",
      "Properties" : {
        "ImageId" : { "Fn::FindInMap" : [ "RegionMap", { "Ref" : "AWS::Region" }, "AMI" ]},
        "InstanceType" : { "Fn::If" : [
          "CreateProdResources",     <<<----If env is prod, instance size is c1.xlarge,
else if env is dev
          "c1.xlarge",
          {"Fn::If" : [
            "CreateDevResources",    <<<----If env is dev, size is m1.large, else use
m1.small
            "m1.large",
            "m1.small"
          ]}
        ]}
      }
    },
    "NewVolume" : {
      "Type" : "AWS::EC2::Volume",
      "Condition" : "CreateProdResources",  <<<----If env is prod, create new volume
      "Properties" : {
        "Size" : "100",
        "AvailabilityZone" : { "Fn::GetAtt" : [ "EC2Instance", "AvailabilityZone" ]}
      }
    }
  }
```

So in this case, you define conditions and then reference the conditions within your resource to make a "decision". For example, if the parameter for environment is equal to "prod", a decision is made to make the instance size c1.xlarge, else if the environment equals "dev", the decision is made to make the instance size m1.large, otherwise default to the instance size m1.small.

You also have the ability of nesting conditions inside other conditions.

```
JSON Nested Example:
"MyAndCondition": {
   "Fn::And": [
      {"Fn::Equals": ["sg-mysggroup", {"Ref": "ASecurityGroup"}]},
      {"Condition": "SomeOtherCondition"}
   ]
}


YAML Nested Example:
MyAndCondition: !And
  - !Equals ["sg-mysggroup", !Ref "ASecurityGroup"]
  - !Condition SomeOtherCondition
```

The condition declared called "MyAndCondition" returns true IF the ASecurityGroup equals a specific value of "sg-mysggroup" AND the condition "SomeOtherCondition" equals true.

So let's talk about this thing called UserData. What's that? Your UserData is 100 lines long and takes hours to troubleshoot if you are missing a space? That means you aren't using it right. The UserData section is a property of an instance that is executed during initial launch. Keyword here is "initial" launch. Write that down, we will talk about it in a bit. Most people like to use the UserData section as their code dumping ground for what you want to run on an instance to install and configure it's services and applications. But what if I told you that most of the code you put in your UserData can be moved into your cloudformation metadata section or an external file in S3?

I know, trippy right? UserData is treated as opaque data so: what you give is what you get back. If you write a line of code to execute that has a syntax error or wouldn't run at the command line of an instance, it's not gonna run here.

Here is an example:

```
JSON UserData Example:
"UserData" : { "Fn::Base64" : { "Fn::Join" : ["", [
          "#!/bin/bash -xe\n",
          "yum install -y aws-cfn-bootstrap\n",

          "# Install the files and packages from the metadata\n",
          "/opt/aws/bin/cfn-init ",
          " --stack ", { "Ref" : "AWS::StackName" },
          " --resource WebServerInstance ",^M
          " --configsets InstallAndRun ",
          " --region ", { "Ref" : "AWS::Region" }, "\n"
]]}}


YAML UserData Example:
UserData:
  Fn::Base64: !Sub |
    #!/bin/bash
    echo "Testing Userdata" > ${FilePath}
    chown ec2-user.ec2-user ${FilePath}
```

Also, UserData is limited to 16KB in raw form, not base64-encoded form and must be base64-encoded before being submitted to the API (hence the "Fn::Base64" function on line 2 and 17). You may also notice the

Join function call for the JSON example on line 2. This takes the comma-delimited list of strings and joins them together as one blob of commands to allow for inputting variables directly into the code such as region and stackname.

Remember when I said to write down that UserData only runs at initial launch? Put that sticky note on your computer monitor. I can't tell you how many times I have been asked why UserData of an instance did not run after a stack update. Ladies and gentlemen, this is because it only runs at INITIAL launch... See what I mean? If you modify your UserData and execute a stack update, you will need to terminate the instance(s) under your auto-scaling group or recreate your EC2 instance before the new UserData will run. I'm glad we got that covered.

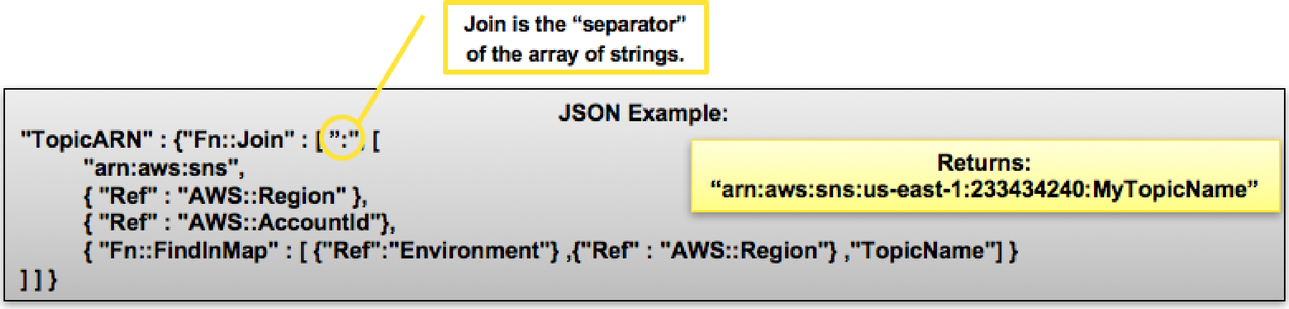
So have you ever heard of this nifty thing called CreationPolicy attributes? Hold onto your seats because this is cool! You can attach a CreationPolicy attribute with a resource "to prevent its status from reaching create complete until AWS CloudFormation receives a specified number of success signals or the timeout period is exceeded". In other words, your stack will not show as being completed until the stack receives a signal or signals within a time window that has been specified under the CreationPolicy attribute. For you visual learners:

```
JSON CreationPolicy Example:
"UserData" : { "Fn::Base64" : { "Fn::Join" : ["", [
             "#!/bin/bash -xe\n",
             "yum update aws-cfn-bootstrap\n",

             "# Install the files and packages from the metadata\n",
             "/opt/aws/bin/cfn-init ",
             "         --stack ", { "Ref" : "AWS::StackName" },
             "         --resource WebServerInstance ",^M
             "         --configsets InstallAndRun ",
             "         --region ", { "Ref" : "AWS::Region" }, "\n",

             "# Signal the status from cfn-init\n",
             "/opt/aws/bin/cfn-signal -e $? ",            <<----Sends signal to stack with
the $? output
             "         --stack ", { "Ref" : "AWS::StackName" },
             "         --resource WebServerInstance ",
             "         --region ", { "Ref" : "AWS::Region" }, "\n"
    ]]}}
},
"CreationPolicy" : {
        "ResourceSignal" : {
          "Timeout" : "PT5M"                  <<--------5 minute timeout if signal is not
received
        }
}
```

The template runs the cfn-signal script to send a success signal with an exit code if all the services are configured and started successfully. If the configuration fails or if the timeout period is exceeded, cfn-signal sends a failure signal that causes the resource creation to fail.

How do I reference something that hasn't been created yet?

So you have a couple options here. First option is to "build" the reference by joining values together. This is used for when you know what the value will be once it has been created and you know that it will exist before the cloudformation script has finished executing. Or if you know the object already exists but don't want to put an ARN as a mapping or parameter value.



Join is the "separator" of the array of strings.

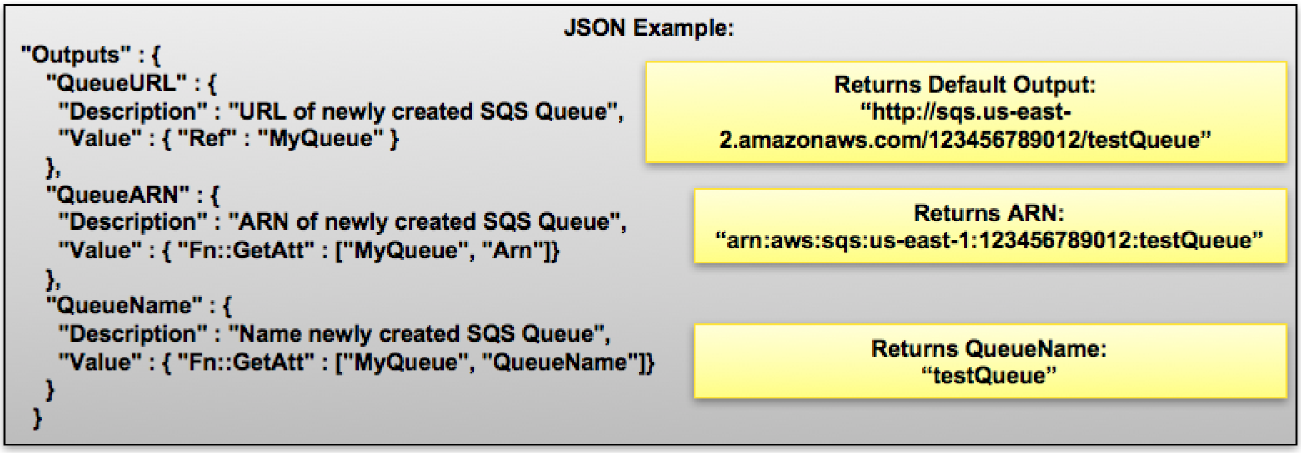
JSON Example:

```
"TopicARN" : {"Fn::Join" : [ ":", [
        "arn:aws:sns",
        { "Ref" : "AWS::Region" },
        { "Ref" : "AWS::AccountId"},
        { "Fn::FindInMap" : [ {"Ref":"Environment"} ,{"Ref" : "AWS::Region"} ,"TopicName"] }
] ] }
```

Returns:
"arn:aws:sns:us-east-1:233434240:MyTopicName"

By using the Join function, you are able to dynamically build the value that you want to reference.

Your second option is to use the output from a different stack or reference an attribute of another resource object within your script.



JSON Example:

```
"Outputs" : {
  "QueueURL" : {
    "Description" : "URL of newly created SQS Queue",
    "Value" : { "Ref" : "MyQueue" }
  },
  "QueueARN" : {
    "Description" : "ARN of newly created SQS Queue",
    "Value" : { "Fn::GetAtt" : ["MyQueue", "Arn"]}
  },
  "QueueName" : {
    "Description" : "Name newly created SQS Queue",
    "Value" : { "Fn::GetAtt" : ["MyQueue", "QueueName"]}
  }
 }
```

Returns Default Output:
"http://sqs.us-east-2.amazonaws.com/123456789012/testQueue"

Returns ARN:
"arn:aws:sqs:us-east-1:123456789012:testQueue"

Returns QueueName:
"testQueue"

Make sure you lookup the default return values of the specific resource because it varies by type. For example, resource type AWS::EC2::Instance has a default return of the instance ID but the default return value of resource type AWS::SQS::Queue is the queue URL. So unless you are feeling lucky, run a quick google of that resource type's return values. By using the GetAtt function like the example above, you can specify exactly what you want as the return value for a resource (i.e. QueueName, Arn, etc.). The attributes you can request are specific to the resource type.

Outputs

Yay we are finally down to the last section. Or you probably just skipped ahead and used your ctrl-f skills. That's okay, I forgive you. The outputs section is exactly what it says, outputs of the CloudFormation stack that you just created. You would configure outputs for various reasons:

- In order to import the value into other stacks (creating cross-stack references)
- As return responses (describe stack calls via AWS CLI). Can be used to assist automation, pipelines, etc.
- Easy identification of values that you don't want to dig in the console for, such as the SQS URL that would require about 5 more clicks.

For whatever reason you wish to configure outputs for, here is an example of how to accomplish that:

```
JSON Example:
"Outputs" : {
  "BackupLoadBalancerDNSName" : {
    "Description": "The DNSName of the backup load balancer",
    "Value" : { "Fn::GetAtt" : [ "BackupLoadBalancer", "DNSName" ]},
    "Condition" : "CreateProdResources"
  },
  "InstanceID" : {
    "Description": "The Instance ID",
    "Value" : { "Ref" : "EC2Instance" }
  }
}


YAML Example:
Outputs:
  BackupLoadBalancerDNSName:
    Description: The DNSName of the backup load balancer
    Value: !GetAtt BackupLoadBalancer.DNSName
    Condition: CreateProdResources
  InstanceID:
    Description: The Instance ID
    Value: !Ref EC2Instance
```

Did I mention conditions can be used in outputs too? No? Well they can. These examples will output two key pairs (of course only if the condition on line 6 and 20 are true, otherwise it will only output one key/pair):

BackupLoadBalancerDNSName	mystack-myelb-15HMABG9ZCN57-1013119603.us-east-1.elb.amazonaws.com
InstanceID	i-636be302

You can view the output values from the console or via AWS CLI. To reference them from a different stack, you would need to configure the output as an "export":

```
JSON Example:
"Outputs" : {
  "StackVPC" : {
    "Description" : "The ID of the VPC",
    "Value" : { "Ref" : "MyVPC" },
    "Export" : {
      "Name" : {"Fn::Sub": "${AWS::StackName}-VPCID" }
    }
  }
}
```

The output key named StackVPC returns the ID of the VPC and exports the value for use from another stack using Fn::ImportValue.

But there is a catch when using the output exports.

Here are the rules when working with exports:

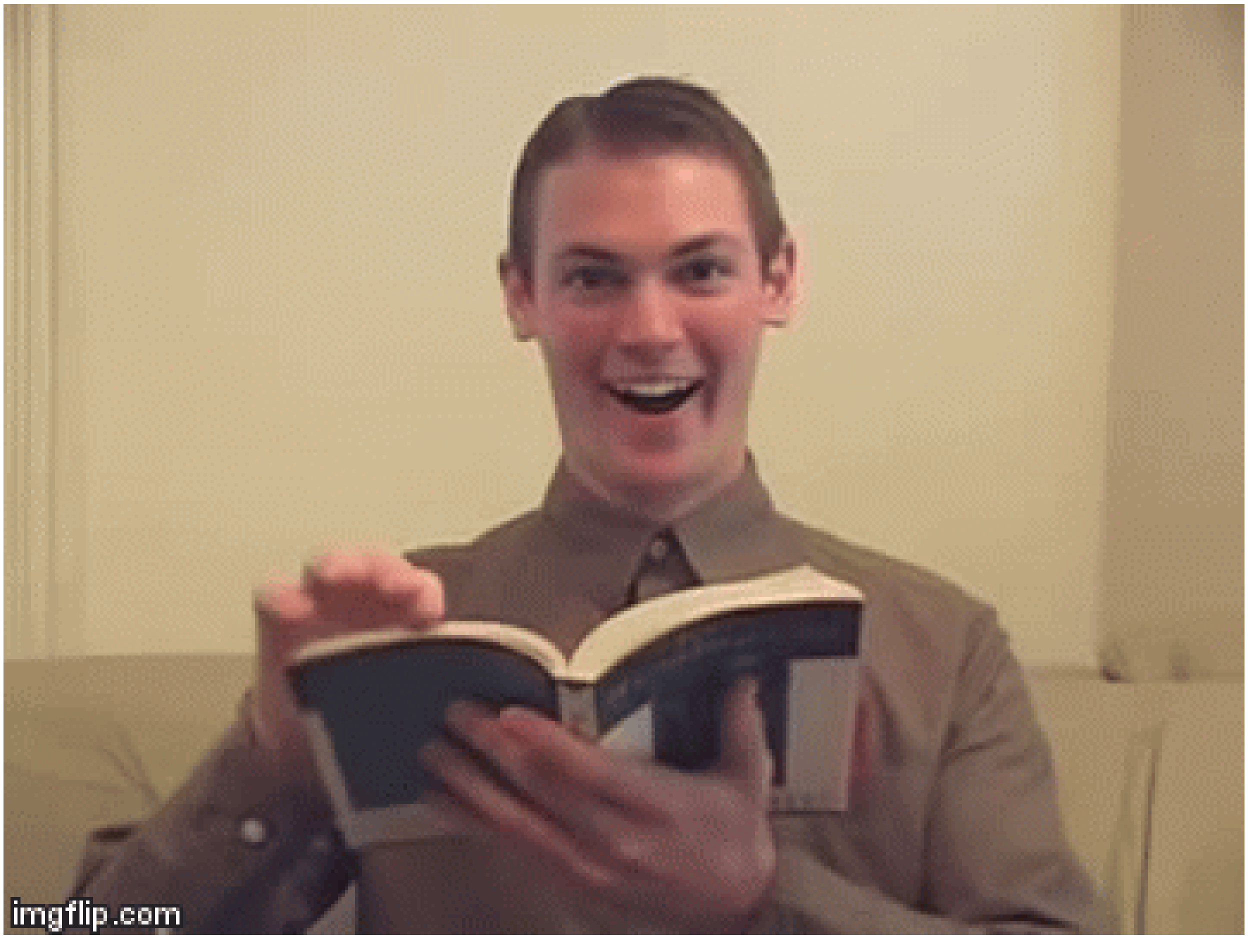
1. You can't delete a stack if another stack references one of its outputs.
2. You can't modify or remove the output value if it's referenced by another stack.
3. You can't create cross-stack references across different regions.
4. Export names must be unique within a region.

So why would anyone use this? Well a use case I could think of is if someone used a CFT template to create multiple CFT stacks and needed to reference values between these stacks such as CFT nesting. But I haven't come up with another use case that would make me want to deal with all the "rules".

Just as a send off, here is some advice to live by:

- The CFT template validator is your friend! In console and CLI!
- Amazon Docs Site is a bottomless pit so before you go down the rabbit hole, have an idea of what you are looking for.
- Have existing resources you manually set up that you want to build a CFT for? Try CloudFormer (template creation beta tool).
- Prefer YAML over JSON? Go for it! It's pretty.
- Use online JSON and YAML validators. They can save you hours of looking for that missing bracket or extra space

Now go off into the interwebz world and research!



549 Views Tags: cloudformation, cft, aws cft, cloudformation templates, aws cloud formation, userdata, aws metadata

Anh Tran *in response to* Dan Coates *on page 16*
Feb 24, 2017 6:19 PM

Great writing and explanation.

Dan, CFT now supports YAML , maybe you like it better that JSON format.

Dan Coates
Dec 16, 2016 2:46 PM

I'm a founding member of the Church of Ansible, so CFTs still make me feel icky inside, but this helped demystify those arcane JSON inscriptions I see all over the Githubs. And your writing style helps get through that level of technical details easily. Thanks!

Jacob T. Massey

Dec 14, 2016 8:05 PM

This is an amazing reference! Thanks for writing this up.