CloudFormation Section 2 Templates-in-Depth

**Template Format and Structure:**

**Template Basics:**

* A template is a declaration of the AWS resources that make up a stack
* Resources are declared in a template
* Resources map to a stack
* Declare object as a name-value pair or a pairing of a name with a set of child objects enclosed
* The resource object is the only required object

**Simple Resource Declaration:**

**JSON**

{

“Resources” : {

“your-bucket-name” : {

“Type” : ”AWS::S3::Bucket

}

}

}

**YAML**

Resources:

Your-bucket-name:

Type:AWS::S3::Bucket

**Template Anatomy:**

* AWSTemplateFormatVersion
* Description- a text string that describes the template
* Matadata- objects that provide additional information about the template
* Parameters- specifies values that you can pass in to your template at runtime
* Mappings- a mapping of keys and associated values that you can use to specify conditional parameter values, similar to lookup tables
* Conditions- defines conditions that control whether certain resources are created or whether certain resource properties are assigned a value during stack creation or update
* Transform- for serverless applications, specifies the version to use
* Resources- specifies the stack resources and their properties
* Outputs- Describes the values that are returned whenever you view your stacks properties

**Template Format and Structure:**

**AWSTemplateFormatVersion:**

* Identifies the capability of the template
* The latest template version is 2010-09-09
* The only valid value currently
* The value for the template format version declaration must be a literal string
* JSON “AWSTemplateFormatVersion” : “2010-09-09”
* YAML: AWSTemplateFormatVersion: “2010-09-09”

**Description:**

* Optional
* Use to describe your template, provide versioning
* “Description” : “Three tier architecture, client xyz, version 1.0.”

**Metadata:**

* Optional
* Include arbitrary JSON or YAML objects that provide details about the template
* Examples would be providing detailed information about a database or a web application you are creating

**Parameters:**

* Pass values into your template when you create a stack
* Each parameter must contain a value when you create a stack
* Bounce out to console – split screen

“Parameters” : {

“InstanceTypeParameter” : {

“Type” : “String”,

“Default” : “t2.micro”,

“AllowedValues” : [“t2.micro”, “m1.small”, “m1.large”],

“Description” : “Enter t2.micro, m1.small, or m1.large. Default is t2.micro.”

}

}

**Mappings:**

* Matches a key to a corresponding set of named values

Mappings:

Mapping01:

Key01:

Name: Value01

Key02:

Name: Value02

**Conditions:**

* Includes statements that define when a resource is created or when a property is defined
* You might use conditions when you want to reuse a template that can create resources in different context
* Example would be for dev environments vs prod environments
* To use conditions, you must include statements in at least 3 different sections of a template: Parameters – the values you want to evaluate, Conditions (Intrinsic Functions), Resource and Outputs (Associate conditions with the resources or outputs that you want to conditionally create)

**Transform:**

* Specifies one or more transforms that CloudFormation uses to process your template
* Simplify templates by condensing template code and enabling reuse of template components
* 2 types: AWS::serverless and AWS::Include
* Lambda
* AWS::Include – very much like includes in popular programming languages, snippets
* Can declare a single transform or multiple transforms within a template

Resources:

MyBucket:

Type: ‘AWS::S3::Bucket”

Properties:

‘Fn::Transform’:

* Name: ‘AWS::Include’

Parameters:

Location: s3:://bucket/mybucketname.yaml

* Name: ‘AWS::Include’

Parameters:

Location: s3://bucket/mybucketacl.yaml

**Resources:**

* Declares the AWS resources that you want to include in the stack

Resources:

Logical ID: This must be unique as it is used in other parts of the template

Type: Resource type identifies the type of resource being created

Properties: Additional options you can specify on your resource

Resources:

MyEC2Instance:

Type: “AWS::EC2::Instance”

Properties:

ImageId: “AMI-####”

**Outputs:**

* Declares output values that you can import into other stacks
* View in the CloudFormation console

Outputs:

Logical ID:

Description: Information about the values

Value: Value to return

Export:

Name: Value to export

**Intrinsic Functions:**

**Intrinsic Functions:**

* Built-in CloudFormation functions which allow you to dynamically assign values to properties at runtime
* Examples: Public IP address of and EC2 instance created in the stack. Get the ip address dynamically during stack creation
* Help to manage CloudFormation stacks
* Enables template ruse and portability
* Intrinsic functions can only be used in specific parts of a template: resource properties, outputs, metadata attributes, and update policy attributes
* Intrinsic functions can be used to conditionally create stack resources

**Fn::Base64-** Sends the encoded data to EC2 instance from UserData property

JSON : {“Fn::Base64” : valueToEncode }

YAML: Fn::Base64: valueToEncode

**FindInMap-** Returns the value of key in 2 level map declared in mappings section

JSON: {“Fn::FindInMap” : [“MapName”, “TopeLevelKey”, “SecondLevelKey”]}

YAML: Fn::FindInMap: [ MapName, TopLevelKey, SecondLevelKey ]

**Fn::GetAtt-** Retrieves the values of an attributes from a resource in the template

* Useful if you need to get the value at runtime
* Examples- getting the public up address of an EC2 instance created in the template. Getting the DNS name of an ELB.

JSON: {“Fn::GetAtt” : [“logicalNameOfResource”, “attributeName”]}

YAML: Fn::GetAtt: [ logicalNameOfResource, attributeName ]

Example: “Fn::GetAtt” : [ “MyELB”, “DNSName” ]

**Fn::GetAzs-** Returns an array that lists Availability Zones for a specified region

* Helps promote portable/reusable templates
* AZs are no the same for every account. Hard coding is not best practice, but hard coding AZs could need change often.

JSON: {“Fn::GetAZs” : {“Ref” : “AWS::Region”}}

YAML: Fn::GetAZs: Ref: “AWS::Region”

**Fn::Join-** Used to append a list of values into a single value separated by provided delimiter

* If a delimiter is the empty string, the set of values are concatenated with no delimiter

JSON: {“Fn::Join” : [“delimiter”, [ comma-delimited list of values ]]}

YAML: Fn::Join: [ delimited, [ comma-delimited list of values ]]

Example (returns “a:b:c”)

!Join [ “:”, [a,b,c]]

**Fn::Select-** Used to retrieve a single value from a given list. This function does not check index value. If given index is out of range then stack may get an error.

JSON: {“Fn::Select” : [ index, listOfObjects ]}

YAML: Fn::Select [ index, listOfObjects ]

Examples (returns 1) because 1 is the 3rd number. Indexing in languages start at 0 so 0 is the 0th place. 5 is in the 1st place. 1 is in the 2nd place.

{“Fn::Select” : [“2”,[“0”, “5”, “1”, “9”, “3” ]] }

Example for Azs

AvailbilityZone: !Select

-0

-Fn::GetAZs !Ref ‘AWS::Region’

**Fn::Split-** Splits a given string by delimiter and returns a list

JSON: {“Fn::Split” : [“Delimiter”, “string”]}

YAML: Fn::Split: [“delimiter”, “string”]

Example

{“Fn::Split” : [“|”, “a|b|c”]}

Returns “a” “b” “c”

**Fn::Sub-** Substitutes the variable in an input string with the specified values.

* Use this function to construct commands or outputs that include values that are not available until you create or update a stack

JSON: {“Fn::Sub”: [ String, { Var1Name: Var1Value, Var2Name: Var2Value}]}

{“Fn::Sub”: [“www.${Domain}”, {“Domain”: {“Ref” : “RootDomainName” }}]}

**Ref-** Used to retrieve the value of given parameter or resource

JSON: {“Ref”: “logicalname”}

YAML: Ref: logicalname

“MyEIP” : {

“Type” : “AWS::EC2::EIP”,

“Properties” : {

“InstanceId” : {“Ref” : “MyEC2Instance”}

}

}

**Intrinsic Functions Continued:**

**Ref:**

* This intrinsic function returns the value of the specified parameter or resource
* Most useful with parameters
* Refer to user input of parameters using the Ref function
* You can also use them for resources

**Fn::GetAtt:**

* Returns the value of an attribute from a resource in the template
* If one resource depends on the value of another resource, but that value is only available once the resource is created then you would use Fn::GetAtt

**Fn::GetAZs:**

* Returns an array that lists Availability zones for a specified region
* The AZs in a region are not the same for each account

**Fn::Select:**

* Returns a single object from a list of objects by index
* Get an array of availability zones with Fn::GetAZs and then you use Fn::Select to select one of the AZs using an index

**Fn::Join:**

* Appends a set of values into a single value, separated by the specified delimiter.
* Use with Fn::GetAZs and Fn::Select

**Condition Functions:**

**Condition Functions:**

* You can use condition functions to conditionally create stack resources.
* Evaluated based on input parameters that you declare when you create or update a stack
* Associate them with resources or resource properties in the Resources and outputs sections of a template
* Define conditions in the conditions section of the template (Exceptions if you use Fn::If)
* You can use Fn::If condition in the metadata attribute, update policy attribute, and property values in the resources section and output sections of a template.
* Use conditions when you want to reuse a template that can create resources in different context, such as a test environment vs a production environment.

**Common use Case:**

* Evaluate EnvironmentType input parameter for either prod or dev. Based on the condition input, tailor your deployment for either environment. Prod will probably have more compute power. Save money in dev.
* Using a condition, you can automate your deployments across multiple environments to cut costs.
* You can only reference other conditions from the Parameters and Mappings sections.
* You cannot reference the logical ID of a resource in a condition.

**Associating a Condition:**

* If you need to conditionally create resources, resource properties, or outputs, you must associate a condition with them.

NewVolume:

Type: “AWS::EC2::Volume”

Condition: CreateProdResources

Properties:

Size: 100

AvailabilityZone: !GetAtt EC2Instance.AvailabilityZone

**Fn::If**

You only need to specify the condition name

Example:

NewVolume:

Type: “AWS::EC2::Volume”

Properties:

Size:

!If [BigGulp, 50, 5]

AvailabilityZone: !GetAtt: Ec2Instance.AvailabilityZone

DeletionPolicy: Snapshot

**Nested Conditions:**

* You can use conditions inside of conditions

ParentCondition: !And

* !Equals [“sg-mysggroup”, !Ref “ASecurityGroup”]
* !Condition ChildCondition

**Fn::And:**

* Returns true if all the conditions are true, otherwise returns false.
* The minimum number of conditions that you can include is 2, and the max is 10

JSON: “Fn::And”: [{condition}, {…}]

YAML: Fn::And [condition]

Or short form

!And [condition]

**Fn::Equals:**

* Compares if 2 values are equal
* Returns true or false

JSON: “Fn::Equals” :[“value\_1”, “value\_2”]

YAML: Fn::Equals: [value\_1, value\_2]

Example

UseProdCondition:

!Equals [!Ref EnvironmentType, prod]

The ! point indicates the YAML short from of a command. It does not indicate a Boolean not.

**Fn::If:**

* Returns one value if the specified condition evaluates to true, and another value if it is false

JSON: “Fn::If”: [condition\_name, value\_if\_true, value\_if\_false]

YAML: Fn::If: [condition\_name, value\_if\_true, value\_if\_false]

Example

SecurityGroups:

!If [CreateNewSecurityGroup, !Ref NewSecurityGroup, !Ref ExistingSecurityGroup]

**Fn::Not:**

* Returns true if the condition evaluates to false, and returns false if the condition evaluates to true.

JSON: “Fn::Not”: [{condition}]

YAML: Fn::Not: [condition]

!Not [condition]

**Fn::Or:**

* Returns true if any of the conditions are true or returns false if none of the conditions are true

YAML: !Or [!Equals [sg-mysggroup, !Ref ASecurityGroup], Condition: SomeOtherCondition]

**Templates to Stacks – S3:**

* Clicking on a resource in the CloudFormation designer will isolated that resource in the code portion
* Options you have
* Delete, duplicate, documentation
* Validate in the top
* You can create stack by the validate check

**A ctr + space will auto populate options while designing code.**

**Template to Stacks – DynamoDB:**

* DynamoDB table with secondary indexes
* Provisioned throughput
* Constraints
* Local secondary indexes
* Global secondary indexes
* Key Schema is a required property

**CloudFormation Designer:**

* Good to validate template periodically to help troubleshoot better

**Using CloudFormer:**

* A beta product in CloudFormation
* A template creation tool
* The template is created from the existing resources in your AWS account
* Select any supported AWS resources that are running in your account, and CloudFormer creates a template in and S3 bucket.
* Not all AWS resources or properties are supported

**Use Cases:**

* You are very good at creating VPCs and 3 tiered web application environments in the management console but new to CloudFormation
* Let CloudFormer create a template for you
* You need a user name and password that creates an EC2 instance

**Template Best Practices:**

* Use input parameters to pass in information whenever you create or update a stack
* Use constraints to validate templates – you can add constraints to parameters such as MinValue, MaxValue. You can insure that parameters entered by the user are valid and will not cause runtime errors.
* Protect sensitive data – Use noecho for passwords and other sensitive input data
* Use CloudTrail! Integrate CloudTrail with CloudFormations to maintain logs of actions performed within CloudFormation. Audit and compliance.
* Tag your resources. Tagging is a best practice for creating resources outside of CloudFormation as well as within.
* Use AWS::CloudFormation::Init to deploy software on EC2 instances – you can install and configure software applications on amazon EC2 instance by using the cfn-init helper script.
* Use the latest helper scripts – include yum install in the user data section of your template
* Validate your template before using it
* Version your templates
* Do not hard code AMI ID – use input parameters and mappings
* Use WaitCondition – a wait condition can be placed with in a template to make AWS CloudFormation pause the creation of the stack and wait for a signal before it continues to create the stack
* Use DependsOn – specify that the creation of a specific resource DependsOn the creation of another resources.
* Cleanup after your stacks – when a stack is deleted, some artifacts such as snapshots and bucket logs may remain. Create scripts for cleanups.
* Use Cloud-init to automate installation of software packages
* Use helper scripts – CloudFormation provides python helper scripts that you can use to install software and start services on EC2 instances in your stack.
* Use intrinsic functions to make your templates reusable and promote automation
* Use nested stacks to simplify and keep the size of your templates manageable. The limit is 200 resource per template.

**CloudFormation and VPCs:**

* There are parts of CloudFormation that launch into the default VPC. CloudFormer does
* CloudFormer is global
* Modify the template used for launching CloudFormer to your VPC
* Include subnet and VPC in the parameters section