**CI/CD/Automation**

**CloudFormation Primer**

* Template Anatomy:
  + Stack – Cloud Formations unit of grouping for infrastructure
  + Template – a JSON document giving CF instructions on how to act and what to create.
  + Stack Policy – IAM style policy statement which governs what can be changed and by who.
  + Parameters – allow the passing of variables into a template via UI, CLI or API
  + Mappings – allow processing of hash’s (arrays of key value pairs) by the cfnTemplate
  + Resources – where your actual resources are declared **(only mandatory section)**
  + Outputs – results from the template
* CloudFormation Capability
  + CFN can have conditional elements to resources, or the whole resource can be conditional.
  + CF can run scripts within instances
  + CF can expand files within instances
  + Each stack has a stack ID – a template that can be applied 1, 10, 100, 1000 times etc
    - Each stack and associated resources are unique
  + The events during stack creation update and deletion can be configured, meaning cloud formation can be as strict or relaxed, as controlling or as messy as you want.
* When & Where
  + Template to deploy bespoke infrastructure rather than doing it manually
  + To create a repeatable patterned environment, for example a wordpress blog + DB if you were running a web hosting business.
  + To run automated testing for CI/VCD environments, create a dedicated, clean environment, inject code, run testing, produce results and delte the test environment; all with no human impact
  + To be able to define an environment once, and have it deployed to any region of AWS without reconfiguration.
  + To manage infrastructure configuration using software development style versioning and testing concepts.
  + KEY CONCEPT: A template should be designed so it is 3qully suitable for 1, 10, 100 or 1000 applications, in one or more regions.

**CloudFormation Structure**

* Parameters
  + Are a way of passing data info a cloud formation templates.
  + Examples: IP address, instance size, names, password, etc
  + Each parameter definition can also have a number of attributes:
    - Type – string/number/list/CommaDelimetedList/AWS Key Pair/AZ
  + Default Value – Becomes the parameter value if none is specified.
  + Allowed Values – One of more values which the parameter can take.
  + Allowed Pattern – Regular expression that defines the format the parameter can take
  + Min & Max Value – For numbers, specify upper and lower limits.
  + In the absence of parameters, CF picks generated names for elements, allowing a single template to be applied repeatedly to an environment. i.e. to create 100’s of S3 buckets
* Resources
  + The section where resources to be configured are specified.
  + A logical ID (friendly name) and a type MUST be specified within the template for each object being created.
  + In addition to the above, a ‘properties’ JSON object associated with each resource configures resource specific elements – some of which may be required, depending on the resource type.
  + Example: You can use the template to create a single S3 bucket, in this example a name is not specified.
    - It is possible to specify the name of the bucket as one of its properties, you don’t have to.
    - Not doing so, highlights one of the benefits of CF, the same template could be applied 5 times to create 5 resources – names can be auto-generated, allowing large scale template re-use.
    - It’s also worth noting, that when the stack is deleted, the resources are also deleted – self cleanup.
* Resource References
  + To combine parameters and resources use the ‘Ref’ function.
  + Within a JSON object, use the ‘Ref’ key and the parameter Object ID as the value. CF will reference the named parameter.
* Outputs
  + Ouputs are a way of displaying results of a stack creation – either for display on the CLI/UI or for the use in parent stacks.
  + A stack can have many outputs, and each output can be a constructed value, parameter references, pseudo parameters or an output from a function such as fn::GetAtt or Ref.
  + Fn:GetAtt is useful as it can provide attributes of resources created in a stack as an output.

Intrinsic Functions & Conditionals

* Intrinsic Functionas available are:
  + Fn::Base64 – provides base 64 encoding
  + Fn::FindInMap – performs mapping lookups
  + Fn::GetAtt – performs advanced reference lookups against non-default values of resources
  + Fn::GetAZs
  + Fn::Join
  + Fn::Select – value selection from list
  + Fn::Ref – references other resources

Intrinsic Functions & Conditionals

* Base64
  + Accepts plain text and coverts to Base64
  + Its useful when other elements in a stack need Base64 input such as EC2 user data
* FindInMap
  + Performs lookups, it accepts a ‘mappings object’ one or two keys, and returns a value.
  + It is essentially a has/dictionary lookup function.
* GetAtt
  + Geta an attribute of an object within the template or a nested template.
  + Ref allows reference to an object for another object.
  + GetAtt allows specific values for an object to be returned i.e. EC2 instance public IP or private IP
* GetAZs
  + Returns a list of availability zones
  + If specified in a region, it returns a list of AZs in that region.
  + If region is omitted, it returns a list of AZs for the region the template is being applied in.
* Select
  + Selects a single object from a list of objects, can be paired with other functions such as GetAZs
* Join
  + It joins values together
  + Format is “Fn::Join”:[“:”,{“a”, “b”, “c”]] returning “a:b:c”
  + A, b, or c in the above example could be outputs from any other intrinsic function, including ref.
  + Function is generally use to construct complex values to be used by other resources, functions , property values.
* Ref
  + Used to reference other objects or parameter values being related in the template, rather than properties being explicitly specified.

CF Stack Updates

* First the stack policy is checked, updates can be prevented.  This is what a stack policy controls.
* By default, the absence of a stack policy allows all updates.
* Once a stack policy is applied it cannot be deleted.
* Once a policy is applied, by default ALL objects are protected, Update:\* is Denied.
* To remove the default DENY protection of an applied stack policy, you need to update the policy with an explicit “Allow” on one/more/all resources.
* A policy is made of JSON.
* Stack Anatomy - Each policy contains JSON policy documents which has the following:
  + Effect – Allow/Deny
  + Resource
  + NotResource
  + Principal – for stack policies is required to be a “\*”
  + Action – this can be “Update:Modify” “Update:Replace” “Update:Delete” or “Update\*”
  + Condition
* An update can impact a resource in 4 possible ways
  + No interruption
  + Some interruption
  + Replacement
  + Delete

CF Nesting

* With stack nesting, a resource can also be a whole STACK, nesting within a parent template.
* Nested stacks can themselves have nested stacks.
* Why do we nest?
  + It allows a potentially huge set of infrastructure to be split over multiple templates.
  + There are size limits to a template/stack. 460k template limit, assuming it’s in S3.
  + 200 resources limit in one template/stack.
  + 100 mappings, 60 parameters, 60 output limit per stack/template.
  + By nesting you can overcome these limits
  + Nesting allows more effective infrastructure-as-code reuse
  + One template is unwieldly – you may not always need every component
  + You can split off infrastructure and or AD
  + Split off SQL
  + Split off SharePoint
  + Maintaining and using them all separate as needed.
* How do we nest?
  + Create a resource of “Type” : “AWS::CloudFormation::Stack”
  + Using TemplateURL we point at the template on S3
  + Using a parameters section, we can pass key value pairs into the template
  + If we don’t pass a parameter the nested default is used
  + If there isn’t a default and we don’t pass anything, the create will fail

CF Wait Condition & Wait Condition Handlers

* DependsOn – A condition that is used for controlling resource creation order within CF.
  + It has limitations because it cannot create a resource if there is a dependency on an unavailable/uncreated resource.
  + DependsOn will wait until the resource is created before allowing the stack to continue.
* Creation Policies, Wait Conditions & Wait Condition Handlers
  + They all do the same task, influence when a resource is marked as completed, delaying until it’s actually ready.
  + Creation Policies can only be used on EC2 instances and Autoscaling Groups (currently)
* Creation Policies
  + Two main components
    - 1. The creation policy definition within the EC2 or Autoscaling resource. The most important elements are: DesiredCapacity and Count
    - 2. The signal configuration within your instance EC2 user-data or launch configuration use-data. When the number of signals is equal or more than the count in the creation policy, the resource is marked as complete.
* Wait Conations – Generally have 4 components
  + They DependOn the resources you are ‘waiting’ on
  + A Handle property reference the above handle
  + They have a responses timeout
  + They have a ‘count’ – if none is specified the default is 1.

CF Custom Resources

* Custom Resource – A resource type within CF that is backed by SNS or Lambda.
* When a stack is created, updated or deleted a SNS notification is sent to a SNS topic containing the event, and a payload. Or in the case of lambda, a lambda function is invoked and passed an event with the same information.

**OpsWorks**

* Recipes & Cookbooks – You state WHAT you want to happen, and leave Chef/OpsWorks to handle the HOW.
* Berkshelf – Allows you to install custom cookbooks from multiple repositories. Feature enables with Chef 11.10.
  + To have a stack install and use custom cookbooks, you must configure the stack to enable custom cookbooks, if it is not already configured.
* Layers
  + An RDS instance can only be associated with one OpsWorks Stack.
  + A stack clone operation DOESN’T copy an existing RDS instance.
* Lifecycle Events
  + Setup, Configure, Deploy, Un-deploy, Shutdown
  + Each layer has its own recipes for each event.
* Create-Deployments command (Runs deployment or stack commands)
  + Request Parameters
    - AppID – Required for app deployments, but not for other deployment commands.
    - Command – A *DeploymentCommand* object that specifies the deployment command and any associated arguments. (**required**)
    - Comment – A user defined comment.
    - Customjson – A string that contains user-defined, custom JSON. It is used to override the corresponding default stack configuration JSON values. The string should be in the following format: "{\"key1\": \"value1\", \"key2\": \"value2\",...}"
    - InstanceIDs – The instance IDs for the deployment targets.
    - LayerIDs – The layer IDs for the deployment targets.
    - StackID – The stack ID. (**required**)
  + Go read AWS documentation

**CodePipeline**

* Definition: A continuous delivery service you can use to model, visualize and automate the steps required to release software. Allows you to automatically Build, Test, Deploy.
* Release Process
  + Source Build Staging Production
* CodePipeline can deploy apps to EC2 instances by using CodeDeploy, EB, ECS or OpsWorks Stacks

**CodeDeploy**

* Definition: A deployment service that automates application deployments to Amazon EC2 instances, on-premises instances, or serverless Lambda functions.
* Application content that can be deployed:
  + code, serverless AWS Lambda functions, web and configuration files, executables, packages, scripts, multimedia files, and so on
* Deployment Types
  + In-Place, Blue/Green
* AppSpec File: The application specification file (AppSpec file) is a YAML-formatted or JSON-formatted file used by AWS CodeDeploy to manage a deployment. File is used to determine:
  + What it should install onto your instances from your application revision in Amazon S3 or GitHub.
  + Which lifecycle event hooks to run in response to deployment lifecycle events.

**CloudSearch**

* Amazon CloudSearch is a managed service in the AWS Cloud that makes it simple and cost-effective to set up, manage, and scale a search solution for your website or application.

**AutoScaling**

* Lifecycle Hooks - Enable you to perform custom actions by pausing instances as an Auto Scaling group launches or terminates them. For example, while your newly launched instance is paused, you could install or configure software on it.
  + After you add lifecycle hooks to your Auto Scaling group, they work as follows:
    1. Responds to scale out events by launching instances and scale in events by terminating instances.
    2. Puts the instance into a wait state (Pending:Wait or Terminating:Wait). The instance is paused until either you continue or the timeout period ends.
    3. You can perform a custom action using one or more of the following options:
       - Define a CloudWatch Events target to invoke a Lambda function when a lifecycle action occurs. The Lambda function is invoked when Amazon EC2 Auto Scaling submits an event for a lifecycle action to CloudWatch Events. The event contains information about the instance that is launching or terminating, and a token that you can use to control the lifecycle action.
       - Define a notification target for the lifecycle hook. Amazon EC2 Auto Scaling sends a message to the notification target. The message contains information about the instance that is launching or terminating, and a token that you can use to control the lifecycle action.
       - Create a script that runs on the instance as the instance starts. The script can control the lifecycle action using the ID of the instance on which it runs.
    4. By default, the instance remains in a **wait state for one hour**, and then the Auto Scaling group continues the launch or terminate process (Pending:Proceed or Terminating:Proceed). If you need more time, you can restart the timeout period by recording a **heartbeat**. If you finish before the timeout period ends, you can complete the lifecycle action, which continues the launch or termination process.
* Instances can be in a Wait state for a finite period of time. The default is 1 hour.
* Max time instance can be in Wait state is 48 hours or 100x the heartbeat timeout, whichever is smaller.
* When an Auto Scaling group launches or terminates an instance due to a simple scaling policy, a [**cooldown**](https://docs.aws.amazon.com/autoscaling/ec2/userguide/Cooldown.html)takes effect. The cooldown period helps ensure that the Auto Scaling group does not launch or terminate more instances than needed.
* At the conclusion of a lifecycle hook, the result is either ABANDON or CONTINUE.

**CloudFormation**

* Creation Policy - Instructs CloudFormation to wait on an instance until CloudFormation receives the specified number of signals. This policy takes effect only when CloudFormation creates the instance.
  + Must be associated with a resource, such as an EC2 instance or an Auto Scaling group.
  + This association is how CloudFormation knows what resource to wait on. In the example policy, the CreationPolicy is associated with an Auto Scaling group. CloudFormation waits on the Auto Scaling group until CloudFormation receives three signals within five minutes. Because the Auto Scaling group’s desired capacity is set to three, the signal count is set to three (one for each instance).
  + If three signals are not received after five minutes, CloudFormation immediately stops the stack creation labels the Auto Scaling group as failed to create, so make sure you specify a timeout period that gives your instances and applications enough time to be deployed.
* Helper Scripts – AWS CloudFormation provides the following Python helper scripts that you can use to install software and start services on an Amazon EC2 instance that you create as part of your stack:
  + [cfn-init](https://docs.aws.amazon.com/AWSCloudFormation/latest/UserGuide/cfn-init.html): Use to retrieve and interpret resource metadata, install packages, create files, and start services.
  + [cfn-signal](https://docs.aws.amazon.com/AWSCloudFormation/latest/UserGuide/cfn-signal.html): Use to signal with a CreationPolicy or WaitCondition, so you can synchronize other resources in the stack when the prerequisite resource or application is ready.
  + [cfn-get-metadata](https://docs.aws.amazon.com/AWSCloudFormation/latest/UserGuide/cfn-get-metadata.html): Use to retrieve metadata for a resource or path to a specific key.
  + [cfn-hup](https://docs.aws.amazon.com/AWSCloudFormation/latest/UserGuide/cfn-hup.html): Use to check for updates to metadata and execute custom hooks when changes are detected.