AWS CloudFormation

AWS CloudFormation is a service that helps you model and set up Amazon Web Services (AWS) resources so that you can spend less time managing those resources and more time focusing on your applications. It provides a simple, efficient way to define cloud infrastructure and manage it as code.

---

1. What is AWS CloudFormation?

AWS CloudFormation allows users to create and provision AWS infrastructure and services in a predictable, automated way. You define the desired state of your infrastructure using a template, which CloudFormation then uses to automatically provision, update, or delete the required resources.

---

2. Key Concepts in AWS CloudFormation

a. Template

A template is a declarative configuration file that defines your AWS resources. It's written in either JSON or YAML and includes sections that describe the resources, parameters, mappings, outputs, and conditions. A template can define a single resource or a full infrastructure stack.

Template Sections:

- **Parameters**: Define inputs that can be passed to customize a stack at runtime (e.g., EC2 instance types, key pairs).
- **Resources**: The actual AWS services and infrastructure (like EC2 instances, S3 buckets) that CloudFormation provisions and manages.
- **Mappings**: Allow you to map keys to specific values, such as regions to specific AMIs.
- **Conditions**: Logical conditions that determine whether certain resources are created or not based on parameter values.
- **Outputs**: Provide information back to the user (e.g., IP addresses, resource names).

b. Stack

A **stack** is the collection of AWS resources defined in a CloudFormation template. When you create, update, or delete a stack, CloudFormation manages the resources as a single unit.

c. StackSets

StackSets allow you to manage stacks across multiple AWS accounts and regions. You can define a template once and deploy it consistently across several environments.

d. Change Sets

A Change Set represents a preview of the changes that CloudFormation will make to your stack before the changes are applied. This allows for validation and understanding of the impact of a stack update before actually executing it.

---

3. CloudFormation Workflow

a. Define a Template

The first step is to define a CloudFormation template in YAML or JSON. The template should specify all resources and configurations that you want AWS to create and manage.

b. Create or Update a Stack

Using the AWS Management Console, AWS CLI, or SDKs, you provide your template to CloudFormation and create a stack. CloudFormation will read the template and provision all specified resources.

- **Stack Creation**: CloudFormation provisions the resources in the defined order, handling dependencies between resources.
- **Stack Updates**: You can update the stack by submitting a modified version of the template or by changing parameter values.

c. Use Change Sets for Updates

Before applying changes to a stack, you can create a Change Set to review the proposed changes. The Change Set details what will be added, modified, or deleted from the stack, allowing you to assess the impact.

d. Monitoring and Rollback

AWS CloudFormation handles the deployment of resources and monitors for any errors. If any issues occur during the creation or update process, CloudFormation will automatically roll back changes to maintain the integrity of the stack.

---

4. CloudFormation Components in Detail

a. Resources

Resources are the core of a CloudFormation template. Resources can include services like EC2 instances, RDS databases, VPCs, S3 buckets, Lambda functions, etc. The `Resources` section of the template is mandatory, and each resource has a logical name and its associated properties.

Example Resource Definition (YAML):

```yaml
Resources:
  MyEC2Instance:
    Type: 'AWS::EC2::Instance'
    Properties:
      InstanceType: t2.micro
      KeyName: my-key
      ImageId: ami-12345678
```

b. Parameters

Parameters allow you to pass in dynamic values during stack creation or updates. This provides flexibility and reusability of templates across different environments.

Example Parameters Section:

```yaml
Parameters:
  InstanceType:
    Type: String
    Default: t2.micro
    Description: EC2 instance type
```

c. Mappings

Mappings allow you to create fixed, hardcoded values within your template, typically used for region-based selections (e.g., AMI mappings).

Example Mappings Section:

```
Mappings:
  RegionMap:
    us-east-1:
      AMI: ami-0abcdef12345
    us-west-2:
      AMI: ami-0fedcba54321
```

d. Outputs

Outputs can be used to return information about resources after stack creation, such as the public DNS of an EC2 instance or the ARN of an S3 bucket.

Example Outputs Section:

```
Outputs:
  InstancePublicIP:
    Description: "Public IP of the EC2 instance"
    Value: !GetAtt MyEC2Instance.PublicIp
```

e. Conditions

Conditions are used to control resource creation. You can define conditions that evaluate whether resources are created based on parameter values.

Example Conditions Section:

```
Conditions:
  CreateProdResources: !Equals [ !Ref EnvType, 'prod' ]
```

---

5. CloudFormation Features and Benefits

a. Infrastructure as Code

CloudFormation enables you to define your entire cloud infrastructure in a template file, allowing for version control, sharing, and reproducibility.

b. Automatic Rollback

In case of failures during resource creation or updates, CloudFormation can automatically roll back the changes to the last stable state, ensuring system reliability.

c. Drift Detection

Drift Detection is a feature that allows you to detect if any resources in your stack have been manually changed outside of CloudFormation, allowing you to bring them back to their defined state.

d. Cross-Account and Cross-Region

Using StackSets, CloudFormation supports provisioning resources across multiple AWS accounts and regions, useful for enterprise-scale deployment scenarios.

---

6. Best Practices for Using CloudFormation

a. Modularize Templates

Break large, complex templates into smaller, more manageable templates. Use nested stacks to divide templates into logical components.

b. Use Parameters and Outputs

Leverage parameters for flexibility and outputs for easier integration with other stacks or systems. For example, use Outputs to share VPC IDs or Subnet IDs across multiple stacks.

c. Manage Stack Updates Carefully

Always create and review Change Sets before applying updates to avoid unintended consequences. Avoid manual changes to resources managed by CloudFormation (out-of-band changes), as this can cause drift.

d. Tagging Resources

Tag your resources using AWS Tags to organize and manage them effectively. Tags can help with cost allocation, automation, and resource management.

e. Handle Stack Deletion with Care

When deleting stacks, make sure to handle any critical resources (like databases or S3 buckets) to prevent data loss. Use the `DeletionPolicy` attribute to retain or backup resources.

Example DeletionPolicy:

```
Resources:
  MyS3Bucket:
    Type: 'AWS::S3::Bucket'
    DeletionPolicy: Retain
```

---

7. AWS CloudFormation Alternatives

While CloudFormation is a powerful tool, there are other infrastructure-as-code (IaC) tools available in the AWS ecosystem and beyond:

- **Terraform**: An open-source tool by HashiCorp, which is cloud-agnostic and can manage multi-cloud environments.
- **AWS CDK (Cloud Development Kit)**: Allows developers to define cloud infrastructure using familiar programming languages such as Python, TypeScript, or JavaScript.

AWS CloudFormation Delete Policy

The **Delete Policy** in AWS CloudFormation is an attribute that you can specify for a resource to control how AWS CloudFormation handles the resource when the stack is deleted. This attribute ensures that certain resources (like databases, S3 buckets, or EBS volumes) are either preserved, backed up, or deleted when a stack is deleted.

There are three main options for the Delete Policy:

1. Retain

- **Behavior**: When you delete the CloudFormation stack, the resource with the "Retain" policy is not deleted. The resource remains intact and is not affected by the stack deletion.
- **Use Case**: Useful when you have critical resources like databases (e.g., RDS or DynamoDB) that you don't want to lose even when the CloudFormation stack is deleted.

Example:

```
Resources:

  MyS3Bucket:

    Type: 'AWS::S3::Bucket'

    DeletionPolicy: Retain
```

2. Snapshot

- **Behavior**: CloudFormation creates a snapshot of the resource before deleting it. This policy is often used with resources like Amazon RDS databases or EBS volumes.
- **Use Case**: Useful when you want to preserve the state of your data before deletion, allowing recovery in the future.

Example:

```
Resources:

  MyDatabase:

    Type: 'AWS::RDS::DBInstance'

    DeletionPolicy: Snapshot
```

3. Delete

- **Behavior**: The resource is deleted when the stack is deleted. This is the default behavior if no Delete Policy is specified.
- **Use Case**: Use this when you want to clean up all resources entirely when the stack is removed.

Example:

```
Resources:

  MyEBSVolume:

    Type: 'AWS::EC2::Volume'

    DeletionPolicy: Delete
```

Summary of Delete Policy Options:

Delete Policy	Action on Stack Deletion	Typical Use Cases
Retain	Leaves the resource in place, not deleted	S3 buckets, critical databases, long-term storage resources
Snapshot	Creates a snapshot before deletion	RDS databases, EBS volumes where you want a backup before deletion
Delete	Deletes the resource	Resources that don’t need to persist after stack deletion

Best Practices:

- Use **Retain** for resources you want to keep, even if the stack is deleted, like databases with critical data.
- Use **Snapshot** for resources where you want a backup before deletion, ensuring that data can be restored if needed.
- Use **Delete** for non-critical, easily reproducible resources to ensure the stack deletion cleans up everything.

AWS CloudFormation Stack Policy

A **Stack Policy** in AWS CloudFormation is a JSON document that defines the update behaviors for stack resources during a stack update operation. It specifies which resources are protected and which can be updated during the stack update process. The policy helps prevent accidental updates to critical resources that could lead to data loss or service disruption.

Key Concepts of Stack Policy

1. **Resource Protection**: The primary purpose of a stack policy is to protect specific resources within a CloudFormation stack from being updated unintentionally. By defining a stack policy, you can specify which resources should remain unchanged during updates, even if changes are part of the new stack template.
2. **Stack Policy Structure**: A stack policy is a JSON document that contains `Statement` objects with `Effect`, `Action`, `Principal`, and `Resource` elements.
    - **Effect**: Defines whether the action is allowed or denied. It can be either `Allow` or `Deny`.
    - **Action**: Specifies the operation to control, such as `Update:Modify`, which controls updates to stack resources.
    - **Resource**: Specifies which resources are affected by the stack policy. This can be all resources (`*`) or specific resources within the stack.
3. **Default Behavior**: If no stack policy is applied, all resources are unprotected and subject to updates. The stack policy allows you to impose selective restrictions on critical resources.

Example of a Stack Policy

Here's an example of a simple stack policy that denies updates to an Amazon RDS database resource but allows updates to all other resources in the stack:

```
{
  "Statement": [
    {
      "Effect": "Deny",
      "Action": "Update:Modify",
      "Principal": "*",
      "Resource": "LogicalResourceId/MyRDSInstance"
    },
    {
      "Effect": "Allow",
      "Action": "Update:*",
```

```
      "Principal": "*",
      "Resource": "*"
    }
  ]
}
```

Explanation of the Example:

- The first statement denies the ability to update the resource with the logical ID `MyRDSInstance` (likely a critical RDS database).
- The second statement allows updates to all other resources in the stack.

Managing Stack Policies

You can apply, modify, or override a stack policy using:

- **AWS Management Console**: By specifying a stack policy when you create or update a stack.
- **AWS CLI or SDK**: Using commands like `set-stack-policy` to apply or update the policy for an existing stack.

Commands for Stack Policies

- **Set a Stack Policy**:

AWS CLI command:

```
aws cloudformation set-stack-policy --stack-name MyStack
--stack-policy-body file://policy.json
```

- **Get Stack Policy**:

AWS CLI command:

```
aws cloudformation get-stack-policy --stack-name MyStack
```

- **Override a Stack Policy During Update**:

AWS CLI command (used when you need to temporarily override the policy for specific updates):

```
aws cloudformation update-stack --stack-name MyStack --template-body
```

```
file://template.json --stack-policy-during-update-body
file://override-policy.json
```

Benefits of Stack Policies

- **Prevent Accidental Updates**: Protect critical resources (e.g., databases, S3 buckets) from being inadvertently updated during stack changes.
- **Controlled Rollouts**: Allows you to define which parts of your infrastructure can be safely updated, reducing the risk of introducing breaking changes.
- **Customizable**: You can update or override the stack policy as needed, providing flexibility for managing your infrastructure.

Best Practices for Stack Policies

- **Use Stack Policies for Critical Resources**: Apply stack policies to resources like databases or resources that store persistent data to prevent accidental updates that could lead to data loss.
- **Override with Caution**: When temporarily overriding stack policies to update protected resources, ensure that you fully understand the impact of the changes.
- **Monitor Stack Updates**: Even with stack policies, it's important to monitor stack updates using AWS CloudWatch or the CloudFormation events tab to verify that changes behave as expected