What are mappings?

- Mappings are fixed variables within your CloudFormation Template.
- They're very handy to differentiate between different environments (dev vs prod), regions (AWS regions), AMI types, etc
- All the values are hardcoded within the template
- Example:

```yaml
Mappings:
  Mapping01:
    Key01:
      Name: Value01
    Key02:
      Name: Value02
    Key03:
      Name: Value03
```

```yaml
Mappings:
  AmiRegionMap:
    us-east-1:
      AMI: ami-0fa1ca9559f1892ec
    us-east-2:
      AMI: ami-09f85f3aaae282910
    us-west-1:
      AMI: ami-080d1454ad4fabd12

Resources:
  MyInstance:
    Type: AWS::EC2::Instance
    Properties:
      ImageId: !FindInMap
        - AmiRegionMap
        - !Ref AWS::Region
        - AMI
      InstanceType: !Ref InstanceTypeParameter
      KeyName: !Ref KeyPair
      Tags:
        - Key : "Name"
          Value: !Join ["-",[uttara,dhaka,bangladesh]]
```

When would you use mappings vs parameters?

- Mappings are great when you know in advance all the values that can be taken and that they can be deduced from variables such as
  - Region
  - Availability Zone
  - AWS Account
  - Environment (dev vs prod)
  - Etc…
- They allow safer control over the template.

- Use parameters when the values are really user specific

Fn::FindInMap Accessing Mapping Values

- We use **`Fn::FindInMap`** to return a named value from a specific key
- **`!FindInMap [ MapName, TopLevelKey, SecondLevelKey ]`**

```
AWSTemplateFormatVersion: "2010-09-09"
Mappings:
  RegionMap:
    us-east-1:
      "32": "ami-6411e20d"
      "64": "ami-7a11e213"
    us-west-1:
      "32": "ami-c9c7978c"
      "64": "ami-cfc7978a"
    eu-west-1:
      "32": "ami-37c2f643"
      "64": "ami-31c2f645"
    ap-southeast-1:
      "32": "ami-66f28c34"
      "64": "ami-60f28c32"
    ap-northeast-1:
      "32": "ami-9c03a89d"
      "64": "ami-a003a8a1"
Resources:
  myEC2Instance:
    Type: "AWS::EC2::Instance"
    Properties:
      ImageId: !FindInMap [RegionMap, !Ref "AWS::Region", 32]
      InstanceType: m1.small
```

What are outputs?

- The Outputs section declares *optional* outputs values that we can import into other stacks (if you export them first)!
- You can also view the outputs in the AWS Console or in using the AWS CLI
- They're very useful for example if you define a network CloudFormation, and output the variables such as VPC ID and your Subnet IDs
- It's the best way to perform some collaboration cross stack, as you let expert handle their own part of the stack
- You can't delete a CloudFormation Stack if its outputs are being referenced by another CloudFormation stack

Outputs Example

- Creating a SSH Security Group as part of one template
- We create an output that references that security group

```yaml
Outputs:
  StackSSHSecurityGroup:
    Description: The SSH Security Group for our Company
    Value: !Ref MyCompanyWideSSHSecurityGroup
    Export:
      Name: SSHSecurityGroup
```

Cross Stack Reference

- create a second template that leverages that security group
- For this, we use the **`Fn::ImportValue`** function
- You can't delete the underlying stack until all the references are deleted too.

```yaml
Resources:
  MySecureInstance:
    Type: AWS::EC2::Instance
    Properties:
      AvailabilityZone: us-east-1a
      ImageId: ami-a4c7edb2
      InstanceType: t2.micro
      SecurityGroups:
        - !ImportValue SSHSecurityGroup
```

What are conditions used for?

- Conditions are used to control the creation of resources or outputs based on a condition.
- Conditions can be whatever you want them to be, but common ones are:
  - Environment (dev / test / prod)
  - AWS Region
  - Any parameter value
- Each condition can reference another condition, parameter value or mapping

How to define a condition?

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

- The logical ID is for you to choose. It's how you name condition
- The intrinsic function (logical) can be any of the following:
  - `Fn::And`
  - `Fn::Equals`
  - `Fn::If`
  - `Fn::Not`
  - `Fn::Or`

Using a condition

- Conditions can be applied to resources / outputs / etc…

```yaml
Resources:
  MountPoint:
    Type: "AWS::EC2::VolumeAttachment"
    Condition: CreateProdResources
```

CloudFormation Intrisic Functions

- Ref
- Fn::GetAtt
- Fn::FindInMap
- Fn::ImportValue
- Fn::Join
- Fn::Sub
- Condition Functions (Fn::If, Fn::Not, Fn::Equals, etc…)

Fn::Ref

- The Fn::Ref function can be leveraged to reference
  - Parameters => returns the value of the parameter
  - Resources => returns the physical ID of the underlying resource (ex: EC2 ID)
- The shorthand for this in YAML is !Ref

```yaml
DbSubnet1:
  Type: AWS::EC2::Subnet
  Properties:
    VpcId: !Ref MyVPC
```

Fn::GetAtt

- Attributes are attached to any resources you create
- To know the attributes of your resources, the best place to look at is the documentation.
- For example: the AZ of an EC2 machine!

```yaml
Resources:
  EC2Instance:
    Type: "AWS::EC2::Instance"
    Properties:
      ImageId: ami-1234567
      InstanceType: t2.micro
```

```yaml
NewVolume:
  Type: "AWS::EC2::Volume"
  Condition: CreateProdResources
  Properties:
    Size: 100
    AvailabilityZone:
      !GetAtt EC2Instance.AvailabilityZone
```

Fn::FindInMap Accessing Mapping Values

- We use **`Fn::FindInMap`** to return a named value from a specific key
- **`!FindInMap [ MapName, TopLevelKey, SecondLevelKey ]`**

```yaml
Mappings:
  AmiRegionMap:
    us-east-1:
      AMI: ami-0fa1ca9559f1892ec
    us-east-2:
      AMI: ami-09f85f3aaae282910
    us-west-1:
      AMI: ami-080d1454ad4fabd12

Resources:
  MyInstance:
    Type: AWS::EC2::Instance
    Properties:
      ImageId: !FindInMap
        - AmiRegionMap
        - !Ref AWS::Region
        - AMI
      InstanceType: !Ref InstanceTypeParameter
      KeyName: !Ref KeyPair
```

Fn::ImportValue

- Import values that are exported in other templates
- For this, we use the **`Fn::ImportValue`** function

```yaml
Resources:
  MySecureInstance:
    Type: AWS::EC2::Instance
    Properties:
      AvailabilityZone: us-east-1a
      ImageId: ami-a4c7edb2
      InstanceType: t2.micro
      SecurityGroups:
        - !ImportValue SSHSecurityGroup
```

Fn::Join

- Join values with a delimiter

```
!Join [ delimiter, [ comma-delimited list of values ] ]
```

- This creates "a:b:c"

```
!Join [ ":", [ a, b, c ] ]
```

Function Fn::Sub

- Fn::Sub, or !Sub as a shorthand, is used to substitute variables from a text. It's a very handy function that will allow you to fully customize your templates.
- For example, you can combine Fn::Sub with References or AWS Pseudo variables!
- String must contain ${VariableName} and will substitute them

```
!Sub
  - String
  - { Var1Name: Var1Value, Var2Name: Var2Value }
```

```
!Sub String
```

The **Fn::Sub** intrinsic function in AWS CloudFormation allows you to substitute variables in an input string with values you specify.

Here's an example of how you can use **Fn::Sub** in a CloudFormation template:

```yaml
AWSTemplateFormatVersion: '2010-09-09'
Description: Example using Fn::Sub in AWS CloudFormation

Parameters:
  MyBucketName:
    Type: String
    Default: my-bucket-name

Resources:
  MyBucket:
    Type: 'AWS::S3::Bucket'
    Properties:
      BucketName:
        Fn::Sub: '${MyBucketName}-${AWS::Region}'  # Using Fn::Sub

Outputs:
  MyBucketNameOutput:
    Value: !Ref MyBucket
```

In this example:

- The CloudFormation template defines a parameter **MyBucketName** that allows you to specify the bucket name.
- The **MyBucket** resource creates an Amazon S3 bucket and uses **Fn::Sub** to concatenate the **MyBucketName** parameter value with the AWS region (**${AWS::Region}**) to form the bucket name.
- The **Outputs** section displays the reference (**!Ref**) to the created S3 bucket.

When you deploy this CloudFormation stack, you can provide a value for the **MyBucketName** parameter. CloudFormation will create an S3 bucket with a name derived from the parameter value and the AWS region using **Fn::Sub**.

This is a simple example, but **Fn::Sub** can be used in various scenarios within CloudFormation templates to dynamically substitute values into strings.

Condition Functions

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

- The logical ID is for you to choose. It's how you name condition
- The intrinsic function (logical) can be any of the following:
  - `Fn::And`
  - `Fn::Equals`
  - `Fn::If`
  - `Fn::Not`
  - `Fn::Or`

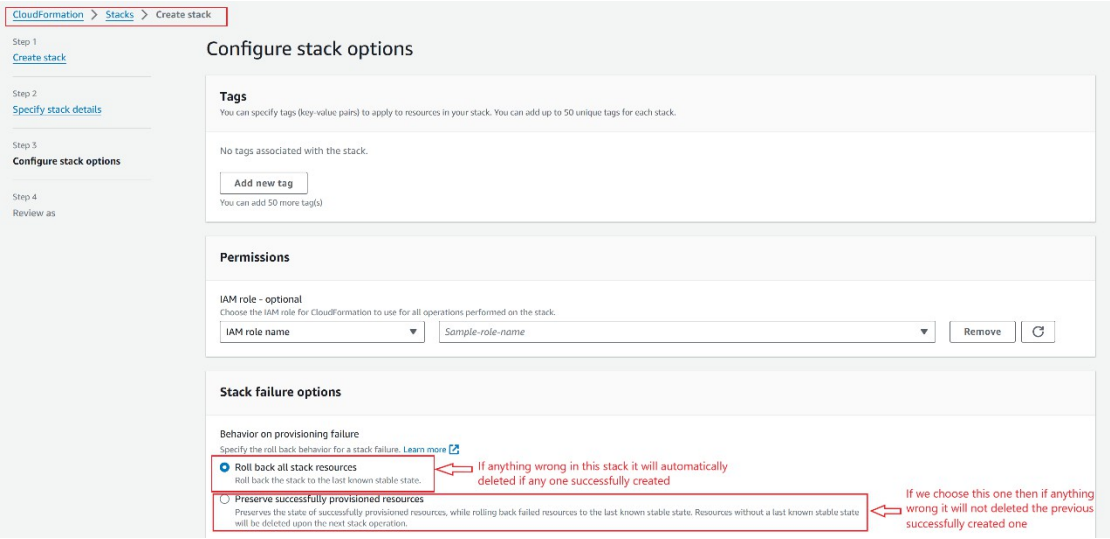
- Conditions can be applied to resources / outputs / etc…

```
Resources:
  MountPoint:
    Type: "AWS::EC2::VolumeAttachment"
    Condition: CreateProdResources
```

CloudFormation Rollbacks

- Stack Creation Fails:
  - Default: everything rolls back (gets deleted). We can look at the log
  - Option to disable rollback and troubleshoot what happened

- Stack Update Fails:
  - The stack automatically rolls back to the previous known working state
  - Ability to see in the log what happened and error messages



Lab: Rollback

create a stack with wrong ec2 image then chose the Roll back all stack resource then create another stack preserve successfully provisioned resources.

CloudFormation Drift

- CloudFormation allows you to create infrastructure
- But it doesn't protect you against manual configuration changes
- How do we know if our resources have **drifted**?

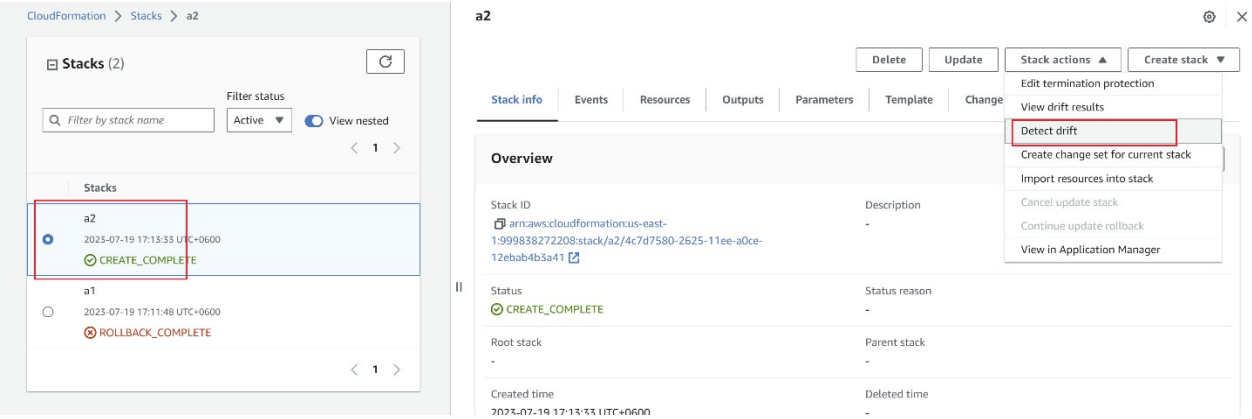
- We can use CloudFormation drift!

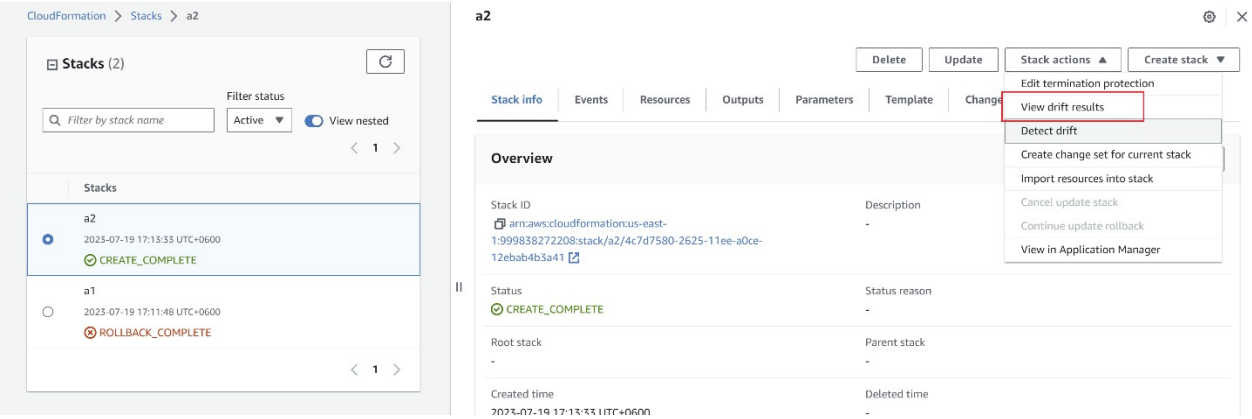
- Not all resources are supported yet: https://docs.aws.amazon.com/AWSCloudFormation/latest/UserGuide/using-cfn-stack-drift-resource-list.html

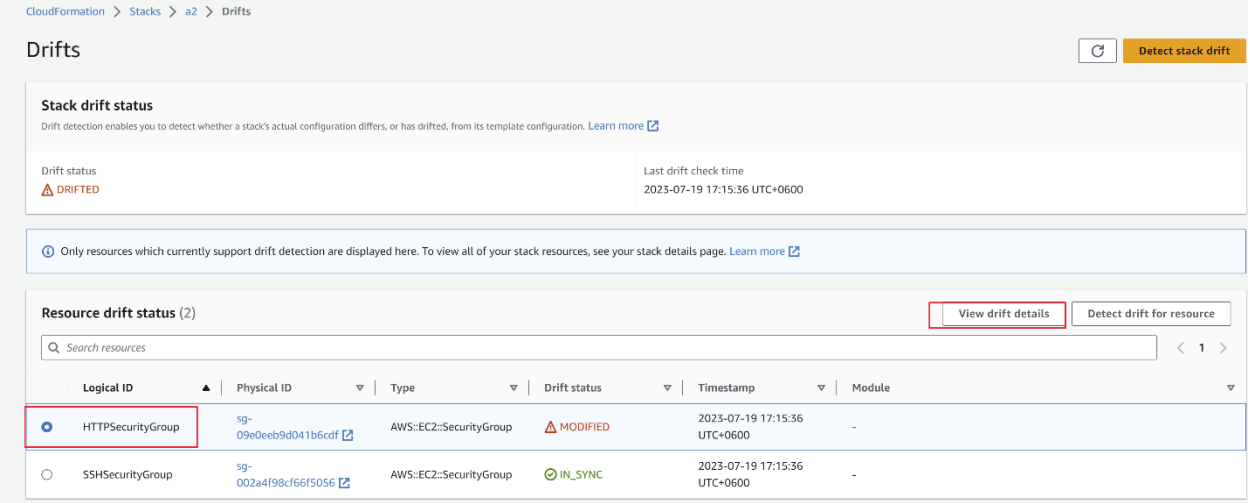
Lab: Drift

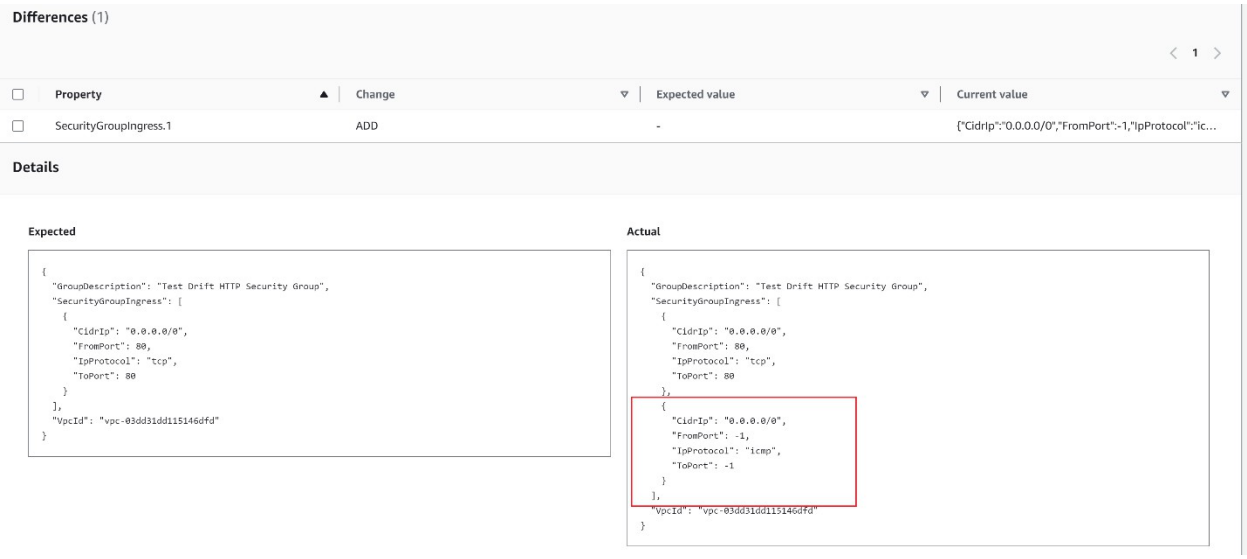
Upload the driff yaml file
go to resource tab and click and security group and edit it for example add icmp for all ip
select the stack > Stack Action >Detect Drift
select the stack > Stack Action >view Drift result.

Now select any modified drift >click View drift details.



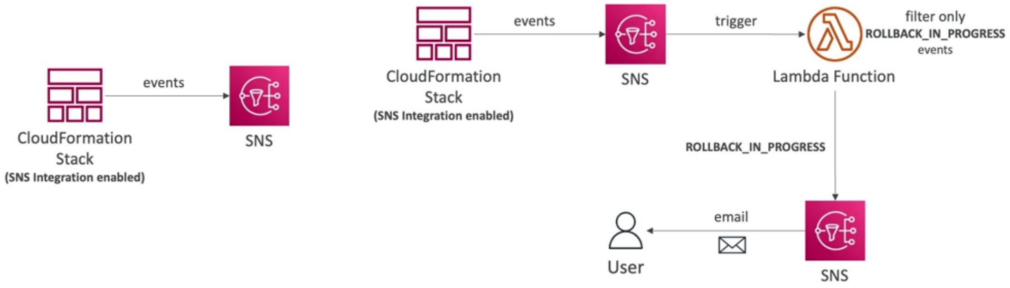






CloudFormation Stack Notifications

- Send Stack events to SNS Topic (Email, Lambda, …)
- Enable SNS Integration using Stack Options



Cross vs Nested Stacks

In AWS CloudFormation, both Cross-Stack References and Nested Stacks are mechanisms used to organize and manage resources within a CloudFormation template.

Cross-Stack References:

Cross-Stack References allow you to export values from one CloudFormation stack and import them into another stack. This functionality enables you to reference resources from different stacks, allowing for greater modularity, reusability, and isolation of resources.

Key points about Cross-Stack References:

- Resources from one stack can be referenced in another stack by exporting their values using the **Export** field in the template.
- These exported values can be imported and used in other stacks by specifying the stack name and the exported value's name.
- Enables loosely coupled architectures by allowing stacks to communicate without tightly coupling resources together.

Nested Stacks:

Nested Stacks enable the creation of a stack within another stack. This mechanism allows for the creation of reusable templates by breaking down complex infrastructure into smaller, manageable units. A parent stack can contain one or more nested stacks, allowing for a modular and hierarchical structure.

Key points about Nested Stacks:

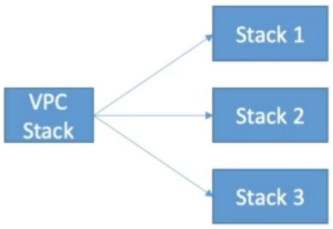
- A main or parent CloudFormation template can call one or more child templates (nested stacks) using the **AWS::CloudFormation::Stack** resource.
- Nested stacks provide a way to create reusable components and manage infrastructure more efficiently.
- Each nested stack operates as an independent stack with its own resources and can be managed separately from the parent stack.

Differences:

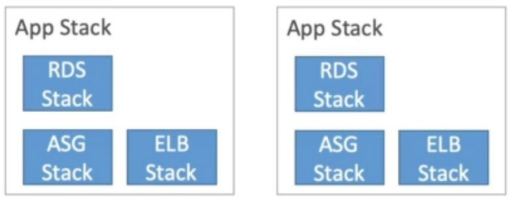
- **Purpose:** Cross-Stack References facilitate communication between different stacks by allowing exported values to be referenced and imported. Nested Stacks aid in breaking down large, complex stacks into smaller, more manageable units.
- **Scope:** Cross-Stack References focus on sharing values (exports/imports) between stacks, while Nested Stacks are about encapsulating sets of resources within a stack.
- **Independence:** Cross-Stack References enable communication between stacks but do not create any dependencies. Nested Stacks are fully contained within a parent stack but operate as separate entities.

In summary, Cross-Stack References facilitate inter-stack communication by exporting and importing values, while Nested Stacks provide a way to create modular and reusable templates within a stack. Both mechanisms serve different purposes but contribute to better organization and management of resources in CloudFormation.

- Cross Stacks
  - Helpful when stacks have different lifecycles
  - Use Outputs Export and Fn::ImportValue
  - When you need to pass export values to many stacks (VPC Id, etc…)



- Nested Stacks
  - Helpful when components must be re-used
  - Ex: re-use how to properly configure an Application Load Balancer
  - The nested stack only is important to the higher level stack (it's not shared)



Nested stacks

AWS CloudFormation supports the concept of nested stacks, which enables you to create a parent-child relationship between stacks. This feature allows you to break down complex CloudFormation templates into smaller, more manageable parts, improving modularity, reusability, and maintainability.

In the context of AWS CloudFormation:

- **Parent Stack:** This refers to the main CloudFormation template that acts as a container for the nested stacks. It includes references to child stacks.
- **Child Stacks:** These are separate CloudFormation templates that are called or referenced within the parent stack. Each child stack represents a subset of resources or infrastructure components.

Benefits of using nested stacks include:

1. **Modularity:** Breaking down large templates into smaller, reusable components enhances the readability and maintainability of CloudFormation templates.
2. **Resource Organization:** It helps organize resources logically by grouping related resources into separate stacks.
3. **Parallel Creation and Deletion:** Nested stacks can be created and deleted in parallel, which can help reduce deployment times for complex architectures.
4. **Reusability:** Common patterns or configurations can be encapsulated into separate stacks and reused across multiple parent stacks or projects.

However, while using nested stacks can be advantageous, there are considerations to keep in mind:

- **Dependency Management:** Be mindful of dependencies between nested stacks. Ensure that the creation and deletion of stacks happen in the correct order to avoid issues with resource dependencies.
- **Template Complexity:** Excessive nesting can introduce complexity, potentially making it more challenging to manage and troubleshoot the infrastructure.

To implement nested stacks, you define the child stack resources within the parent stack using the **AWS::CloudFormation::Stack** resource type. The child stack's template is specified as a parameter within the parent stack's template.

Here's a simplified example of how a nested stack might look within a CloudFormation template:

```yaml
Resources:
  MyNestedStack:
    Type: AWS::CloudFormation::Stack
    Properties:
      TemplateURL: 'URL_of_child_stack_template'
      Parameters:
        # Parameters passed to the child stack
        Param1: 'Value1'
        Param2: 'Value2'
      Tags:
        - Key: Name
          Value: MyNestedStack
```

This **MyNestedStack** resource represents a child stack, and the **TemplateURL** property specifies the location of the child stack's template. Parameters can be passed to the child stack as needed.

By leveraging nested stacks, you can create a more organized and modular infrastructure using AWS CloudFormation.

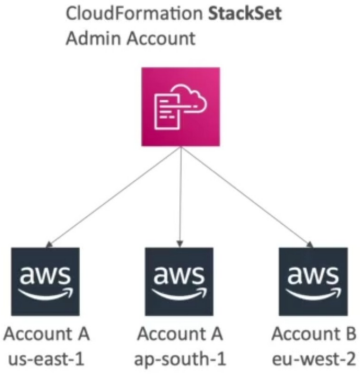
- Nested stacks are stacks as part of other stacks
- They allow you to isolate repeated patterns / common components in separate stacks and call them from other stacks
- Example:
  - Load Balancer configuration that is re-used
  - Security Group that is re-used
- Nested stacks are considered best practice
- To update a nested stack, always update the parent (root stack)

Lab: Nested Stack

Create a stack by using a file (7-nestedstacks.yaml) and we can see we are getting already created stack use in our stack.

StacksSet

- Create, update, or delete stacks across **multiple accounts and regions** with a single operation
- Administrator account to create StackSets
- Trusted accounts to create, update, delete stack instances from StackSets
- When you update a stack set, *all* associated stack instances are updated throughout all accounts and regions.



CloudFormation Stack Polices

- During a CloudFormation Stack update, all update actions are allowed on all resources (default)
- A Stack Policy is a JSON document that defines the update actions that are allowed on specific resources during Stack updates
- Protect resources from unintentional updates
- When you set a Stack Policy, all resources in the Stack are protected by default
- Specify an explicit ALLOW for the resources you want to be allowed to be updated

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

Allow updates on all resources **except** the ProductionDatabase

ChangeSets

AWS CloudFormation Change Sets are a feature that allows you to preview and review proposed changes to a stack before executing those changes. They enable you to see the differences between a current stack configuration and a proposed stack configuration without actually making any changes to the resources.

Key features and purposes of Change Sets in AWS CloudFormation include:

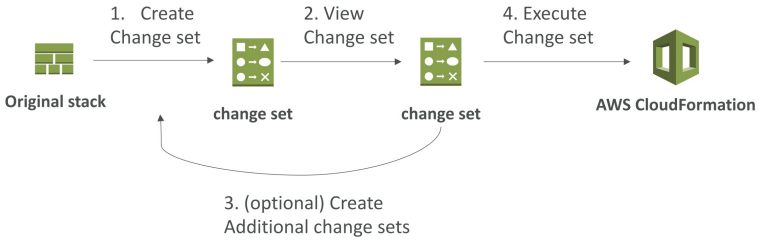
1. **Preview Changes:** When you want to update a CloudFormation stack, you can create a Change Set to see how the changes will impact your existing resources before implementing them.
2. **Review Impact:** Change Sets provide detailed information about what resources will be added, modified, or removed based on the proposed changes.
3. **Granular Control:** They allow you to review individual resource changes, understand dependency changes, and take action accordingly.
4. **Safety Measure:** Change Sets act as a safety mechanism, preventing unintended modifications or deletions by giving you a chance to review and approve changes before executing them.
5. **Visualization:** They help visualize the impact of changes through a summary of additions, modifications, and deletions in the stack's resources.

To create and use a Change Set in AWS CloudFormation:

1. **Create Change Set:** You initiate the creation of a Change Set using the AWS Management Console, AWS CLI, or SDK/APIs. This involves specifying the stack you want to update and the proposed changes.
2. **Review Changes:** Once the Change Set is created, you can review the changes via the AWS Management Console or programmatically using the CLI/SDK. The Change Set provides detailed information about the resources affected by the proposed changes.
3. **Execute or Discard Changes:** After reviewing the changes, you have the option to execute the Change Set to apply the proposed changes to the stack or discard the Change Set if the changes are not as desired.
4. **Merge or Update Changes:** You can also modify the proposed changes within the Change Set before execution, providing flexibility in making adjustments.

Change Sets are particularly useful when you want to assess the impact of changes to your infrastructure, ensure compliance with your expectations, and avoid unintended consequences before actually applying modifications to your AWS CloudFormation stack. They provide a controlled way to manage updates and alterations to your cloud resources.

- When you update a stack, you need to know what changes before it happens for greater confidence
- ChangeSets won't say if the update will be successful



More information:

https://docs.aws.amazon.com/AWSCloudFormation/latest/UserGuide/using-cfn-updating-stacks-changesets.html