# CloudFormation

- Currently, we have been doing a lot of manual work.
- All this manual work will be very tough to reproduce:
  - In another region
  - In another AWS account
  - Within the same region if everything was deleted.
- Would it be a great if we had a code that would deploy everything in a code?
- That code would be deploy and create / update / delete our infrastructure.

## What is CloudFormation?

- CloudFormation is a **`declarative way`** of outlining your AWS infrastructure for any resources ( most of them are supported).
- For example, within a CloudFormation template, you say:
  - ☑ I want a security group
  - ☑ I want two EC2 instances using the Security Groups
  - ☑ I want two EIP s for the EC2 machines
  - ☑ I want an S3 bucket
  - ☑ I want a load Balancer (ELB) in front of these machines.
- Then CloudFormation creates those for you, in the **`right order,`** with the **`exact configuration`** that you specify.

## Benefits of CloudFormation

- **`Infrastructure as a Code:`**
  - ☑ No resource are manually created, which is excellent for control.
  - ☑ The code can be version controlled for example using git.
  - ☑ Changes to the infrastructure are reviewed through code.
- **`Cost:`**
  - ☑ Each resource within the stack is staged with an identifier so you can easily see how much a stack costs you.
  - ☑ You can estimate the cost of your resources using CloudFormation template.
  - ☑ Savings strategy: In Dev, you can make an automation to delete the template at 5 pm and recreate it at 8 AM safely.
- **Productivity:**
  - ☑ Ability to destroy and re-create an infrastructure on the cloud on the fly.
  - ☑ Automated generation of Diagram for your templates!
  - ☑ Declarative programming ( no need to figure out ordering and orchestration)

- **Separation of concern: create many stacks for many apps, and many layers. Example:**
  - ☑ VPC stack
  - ☑ Network Stacks
  - ☑ App stacks

- **Do not re-invent the wheels:**
  - ☑ Leverage existing templates on the web!
  - ☑ Leverage the documentation.

## How CloudFormation Works

- Templates has **to be uploaded in S3** and t**hen referenced** in the CloudFormation.
- To **update a template**, we can not edit previous ones. We have to **re-upload a new version of the template** to AWS
- Stacks are identified by a name
- Deleting a stack deletes every single artifact that was created by CloudFormation.

## Deploying the CloudFormation templates:

1. **Manual way:**
- Editing templates in the CloudFormation Designer.
- Using the console to input parameters, etc

2. **Automated way:**
- Editing templates in a YAML file
- Using the AWS CLI (Command Line Interface) to deploy the templates.
- Recommended way when you fully want to automate your flow

## CloudFormation Building Blocks

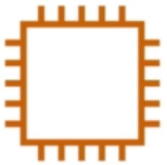
### Template components

1. **RESOURCES:** your AWS resources declared in the template (Mandatory)
2. **PARAMETERS:** the dynamic inputs for your template
3. **MAPPINGS:** the static variables for your templates
4. **OUTPUTS:** Reference to what has been created
5. **CONDITIONALS:** List fo conditions to preform resource creation
6. **METADATA:**

### Template helpers:

1. **Reference**
2. **Functions**

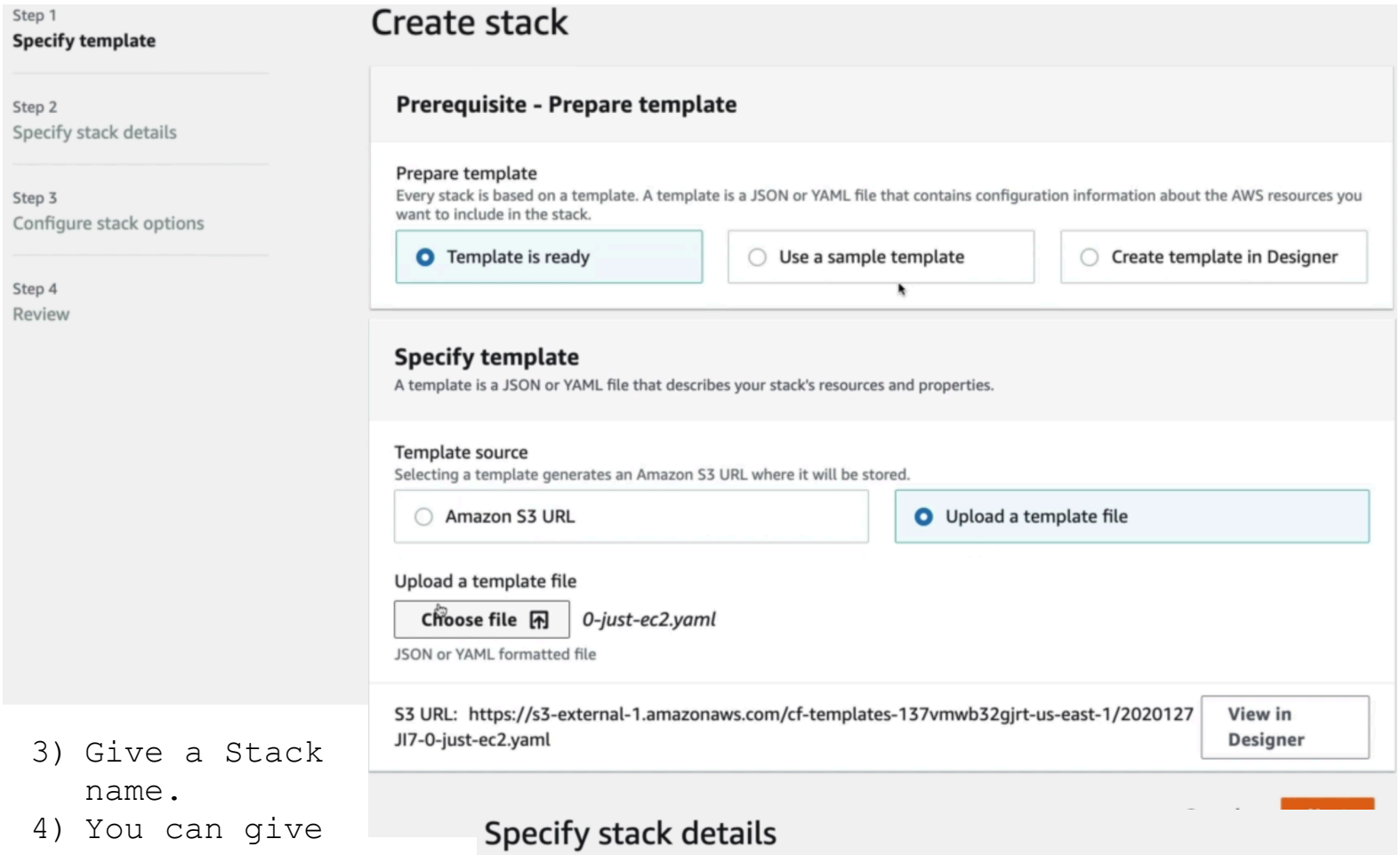
# Introduction to CF.



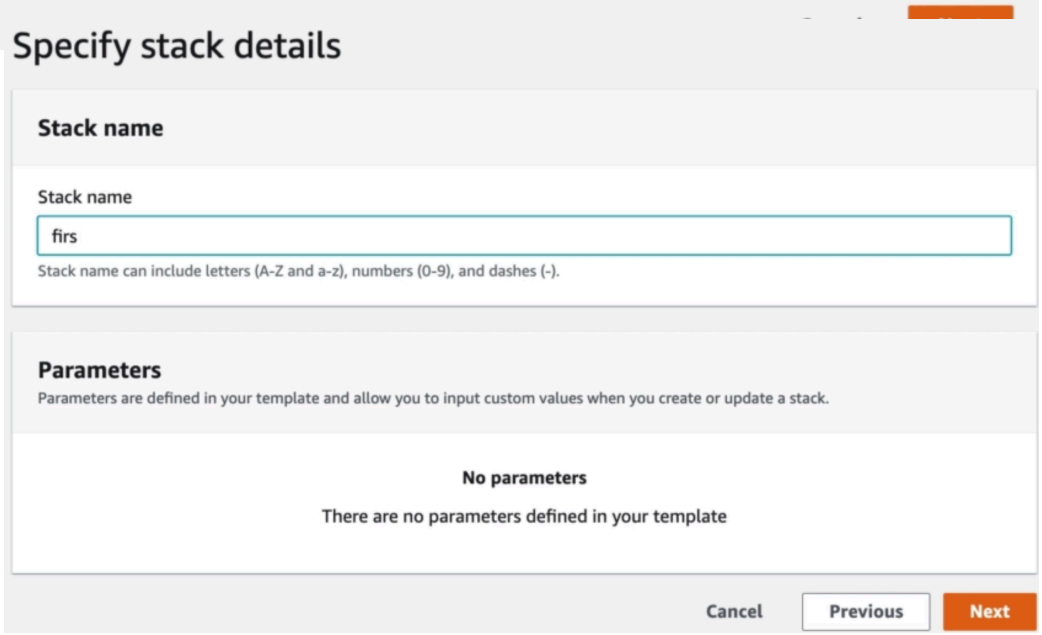
EC2 Instance

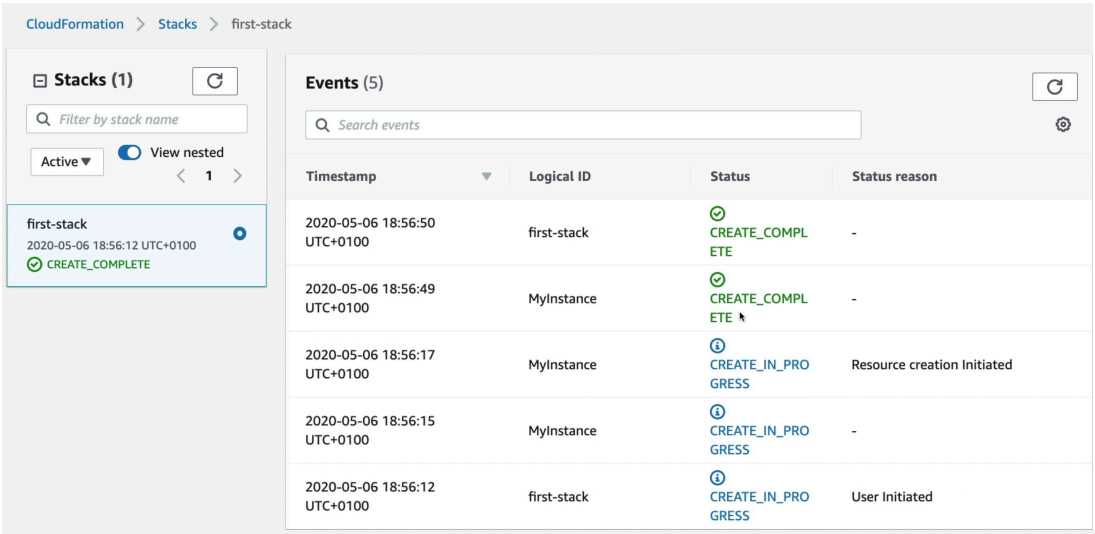
We are going to create a simple EC2 instance.
Then we are going to create to add an EIP to it.
We are going to add two Security Groups to it.

1) Go to the CloudFormation and press Create Stack. This will be our first stack.
2) Our template is ready in the file we will just go and upload that file.Next

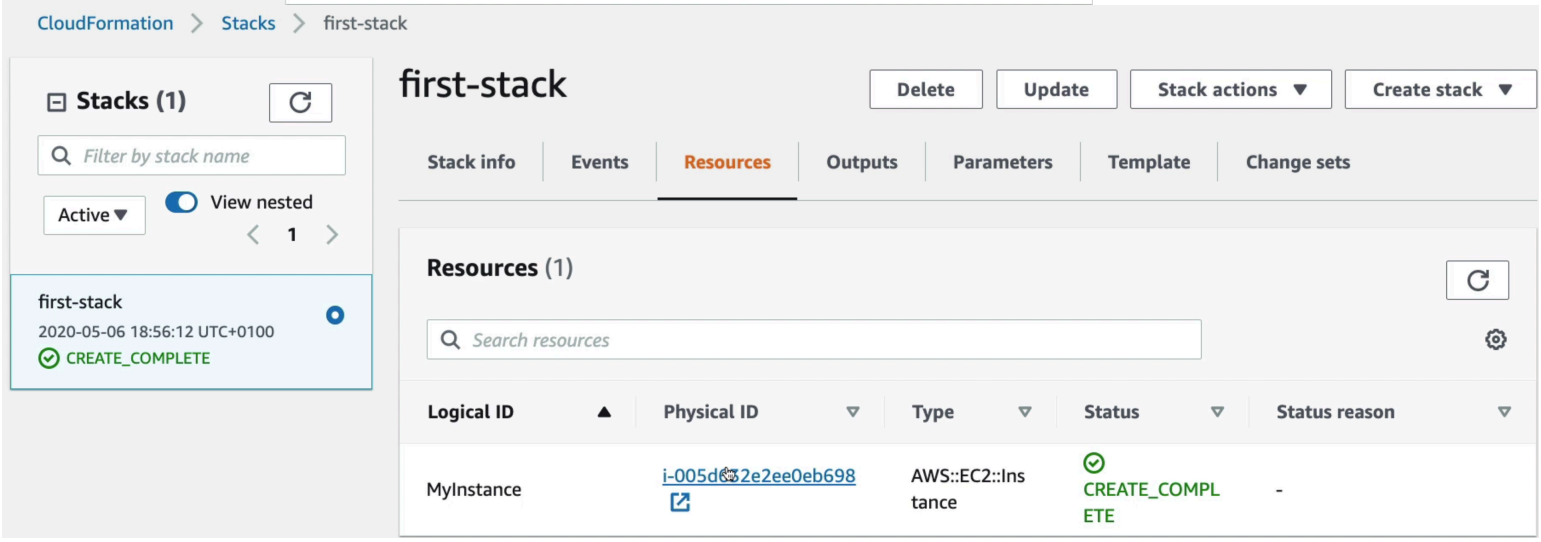



3) Give a Stack name.
4) You can give a stack name and press create for now.

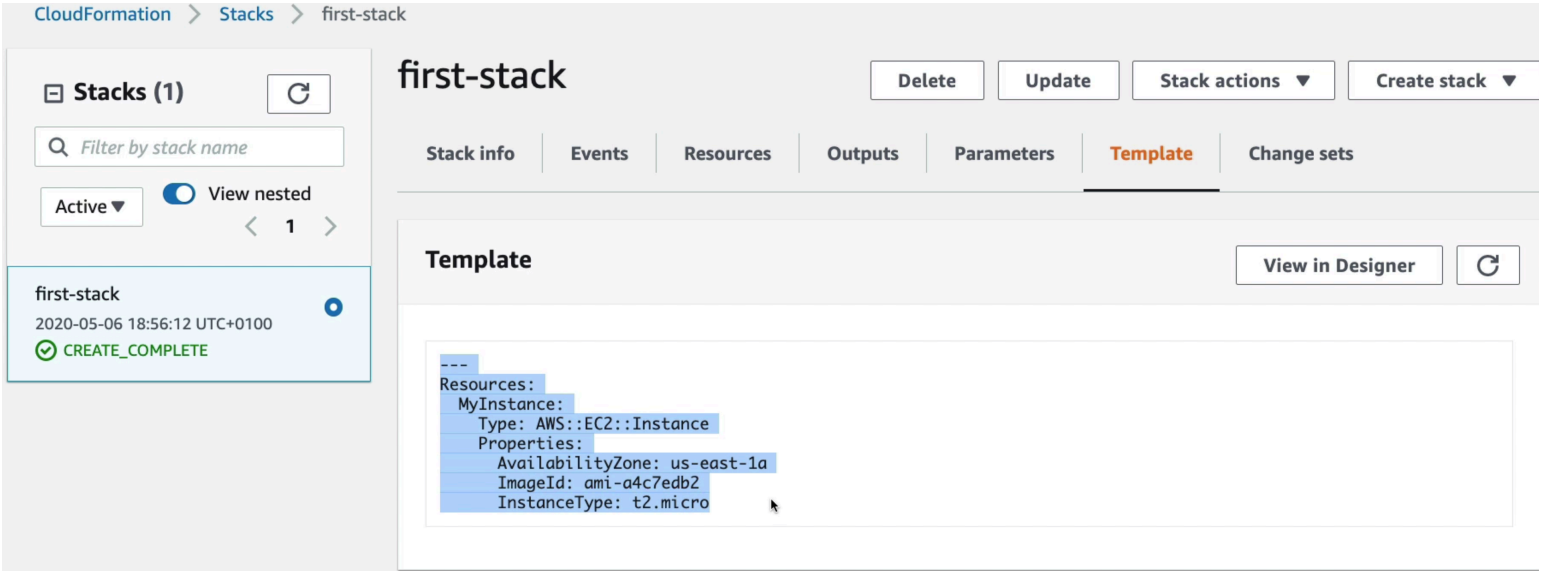





In the **"Events"** section - You can see all the progress what you have been done so far.




**The "Resource" section will show what kind of resources has been created so far.**




**Will show us simple "Template" file for what we have created so far.**

So far this is a vary simple setup and we have created the simple EC2 instance.

# CloudFormation Update and Delete Stack

So in the above we have created an EC2 instance simple template in YAML file. But what we are going to do is we will **update the stack** with the different template.

0-just-ec2.yaml

cloudformation > ! 0-just-ec2.yaml > {} Resources > {} MyInstance > {} Properties > ImageId

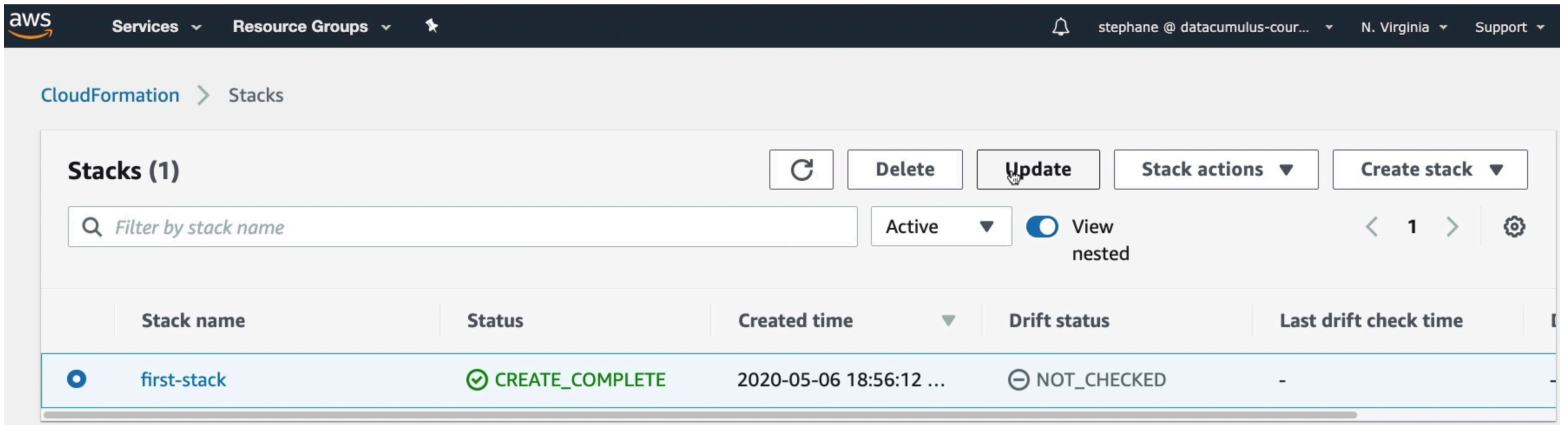
```yaml
---
Resources:
  MyInstance:
    Type: AWS::EC2::Instance
    Properties:
      AvailabilityZone: us-east-1a
      ImageId: ami-a4c7edb2
      InstanceType: t2.micro

```

On the second template what we will do is.
We will add
- EIP
- Two SG
- And EC2

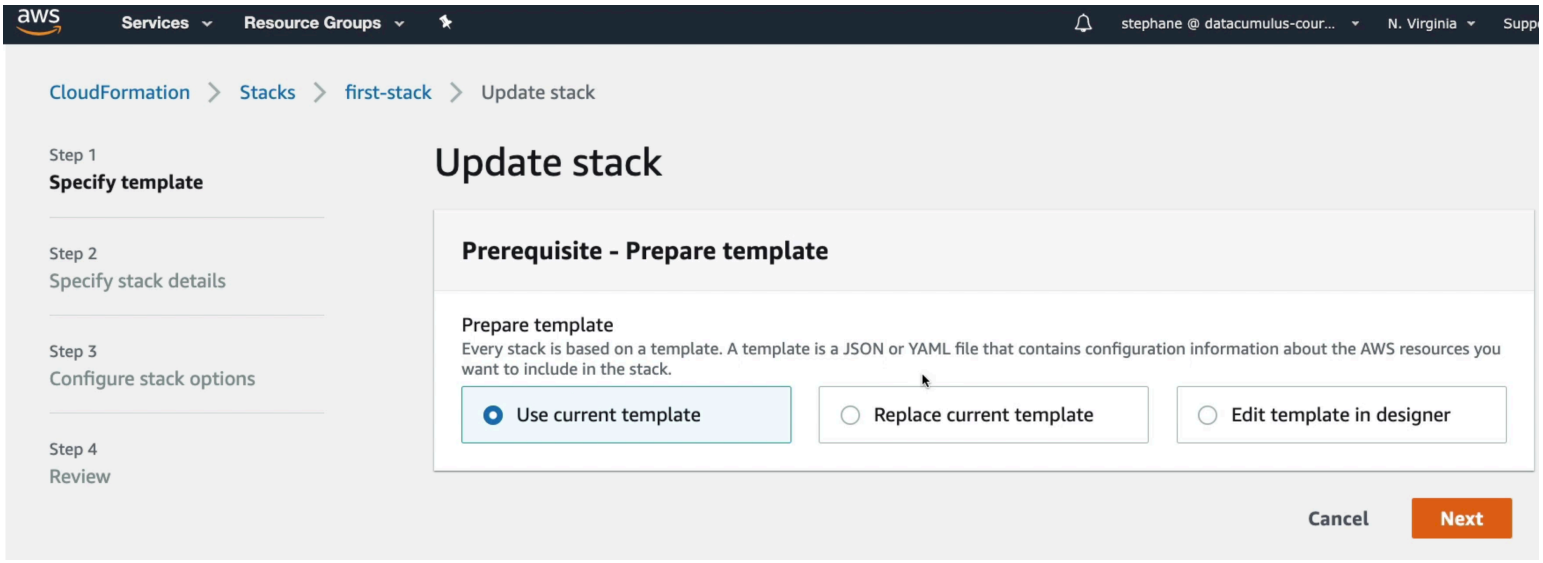
1-ec2-with-sg-eip.yaml

1-ec2-with-sg-eip.yaml > {} Resources > {} ServerSecurityGroup > {} Properties > [ ] SecurityGroupIngress > {} 0 > # ToPo

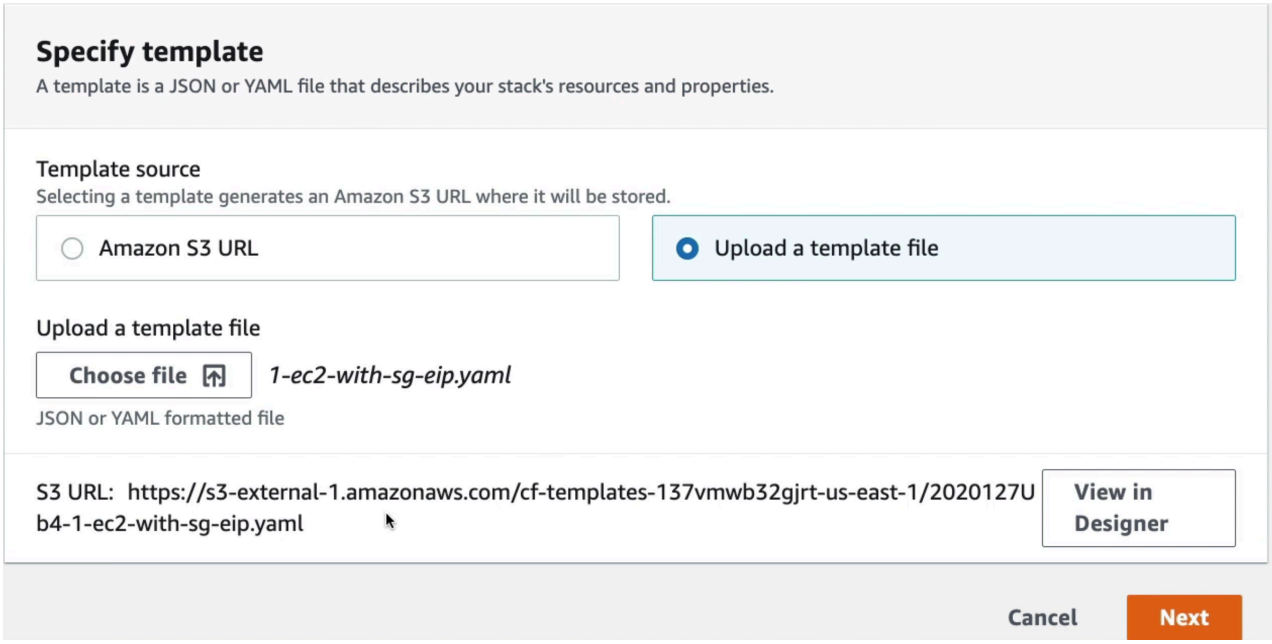
```yaml
---
Parameters:
  SecurityGroupDescription:
    Description: Security Group Description
    Type: String

Resources:
  MyInstance:
    Type: AWS::EC2::Instance
    Properties:
      AvailabilityZone: us-east-1a
      ImageId: ami-a4c7edb2
      InstanceType: t2.micro
      SecurityGroups:
        - !Ref SSHSecurityGroup
        - !Ref ServerSecurityGroup

  # an elastic IP for our instance
  MyEIP:
    Type: AWS::EC2::EIP
    Properties:
      InstanceId: !Ref MyInstance

  # our EC2 security group
  SSHSecurityGroup:
    Type: AWS::EC2::SecurityGroup
    Properties:
      GroupDescription: Enable SSH access via port 22
      SecurityGroupIngress:
      - CidrIp: 0.0.0.0/0
        FromPort: 22
        IpProtocol: tcp
        ToPort: 22

  # our second EC2 security group
  ServerSecurityGroup:
    Type: AWS::EC2::SecurityGroup
    Properties:
      GroupDescription: !Ref SecurityGroupDescription
      SecurityGroupIngress:
      - IpProtocol: tcp
        FromPort: 80
        ToPort: 80
        CidrIp: 0.0.0.0/0
      - IpProtocol: tcp
        FromPort: 22
        ToPort: 22
        CidrIp: 192.168.1.1/32
```



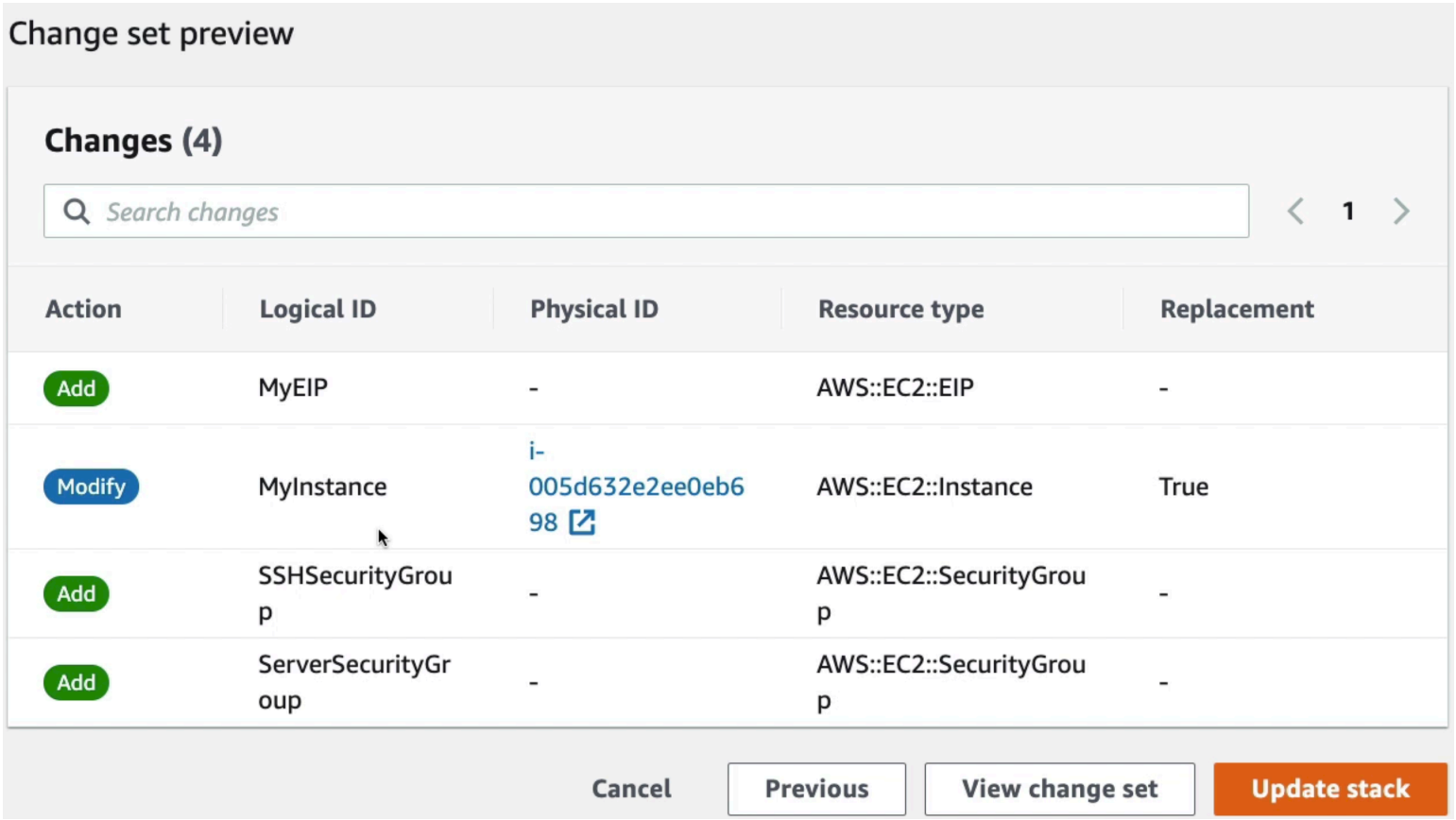

**1) To Update a Stack - you need to go back and select the stack and press “update”**




**2) There will be a 3 options and we will choose to “Replace current template”**



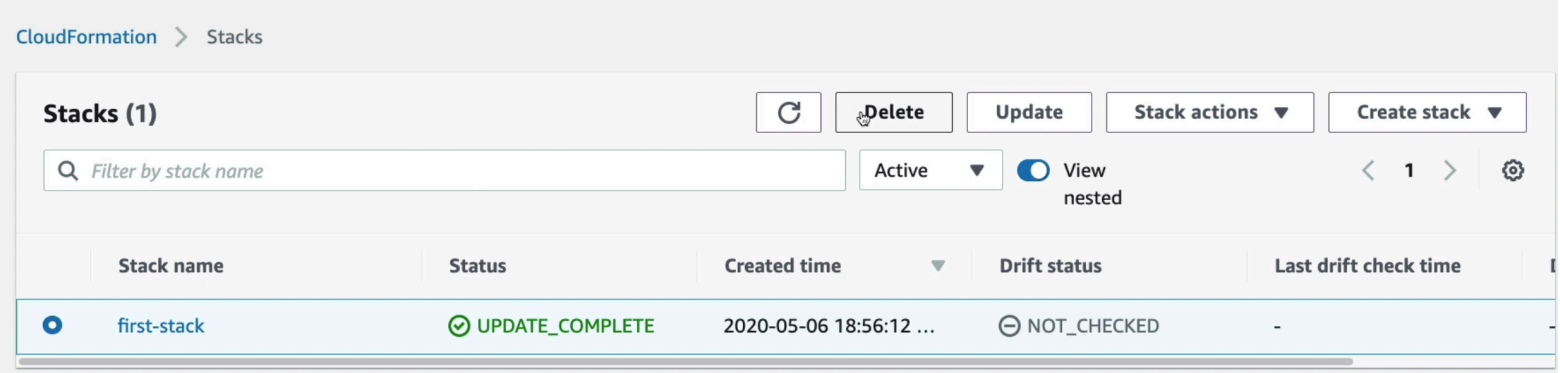

We will upload a new file and as you can see the new file has been uploaded to the S3 but we choose our file and press Next.




In the Review section we get the all the changes that will be done to our existing template. For the Update and it will show us what changes will be done to our template.
Press Update Stack.And as we expected our resources has been created. And it will update our previous stack.

## Deletion Process:

You can either delete manually every resource one by one or what you can do is the right way is delete the stack itself. And it will delete the whole resources that has been created within the stack. And CloudFormation knows what to delete in the right order.



# YAML Crash Course

```yaml
invoice:      34843
date   :      2001-01-23
bill-to:
    given  :  Chris
    family :  Dumars
    address:
        lines: |
            458 Walkman Dr.
            Suite #292
        city    : Royal Oak
        state   : MI
        postal  : 48046
product:
    - sku         : BL394D
      quantity    : 4
      description : Basketball
      price       : 450.00
    - sku         : BL4438H
      quantity    : 1
      description : Super Hoop
      price       : 2392.00
```

CloudFormation uses both YAML and JSON formats but we will go through the YAML in our case because it is little bit easier to use and understand.

**Key value Pairs** - invoice : 34843 line1

**Support Arrays** - line 14 usually it starts with "-" sign sku.

**Nested Objects** - line3 and within it we have nested objects like line4,5,6, and within line 6 we have nested objects line line 7,10 etc

**Multi line Strings** - line 7 "|" is the multi line support.

So far we have created the EC2 instance in the YAML format.

# PARAMETERS

- Parameters are a way to provide inputs to your AWS CloudFormation template.
- Why you need to define parameters:
  - ☑ You want to **reuse** your templates across the company
  - ☑ Some inputs can not be determined ahead of time
- Parameters are extremely powerful, controlled and can prevent errors from happening in your templates **thanks to types**.

### When you should use the Parameters?

You can use Parameters if the CloudFormation resource configuration likely to change in the future? If yes then use parameters.
You don't have to re-upload the template to change it is content.

```yaml
Parameters:
  SecurityGroupDescription:
    Description: Security Group Description
    (Simple parameter)
    Type: String
```

# Parameters Settings

Parameters can be controlled by all these settings:

- Type:
  - String
  - Number
  - CommaDelimitedList
  - List<Type>
  - AWS Parameter (to help catch invalid values – match against existing values in the AWS Account)
- Description
- Constraints
- ConstraintDescription (String)
- Min/MaxLength
- Min/MaxValue
- Defaults
- AllowedValues (array)
- AllowedPattern (regexp)
- NoEcho (Boolean)

## How to Reference a Parameter

- ☑ The **`Fn::Ref`** function can be leveraged to reference parameters.
- ☑ Parameters can be used anywhere in a template
- ☑ The short stand for this in YAML is **`!Ref`**
- ☑ The function can also reference other elements within the template.

```
! 0-just-ec2.yaml    ! 1-ec2-with-sg-eip.yaml ×
1   ---
2   Parameters:
3     SecurityGroupDescription:
4       Description: Security Group Description
5       Type: String
35    # our second EC2 security group
36    ServerSecurityGroup:
37      Type: AWS::EC2::SecurityGroup
38      Properties:
39        GroupDescription: !Ref SecurityGroupDescription
40        SecurityGroupIngress:
41        - IpProtocol: tcp
42          FromPort: 80
43          ToPort: 80
44          CidrIp: 0.0.0.0/0
45        - IpProtocol: tcp
46          FromPort: 22
47          ToPort: 22
48          CidrIp: 192.168.1.1/32
```

In the Example below as you can see we, defined our parameters Line2. It is simple description parameter. And in the line39 we Reference it to the Parameter using !Ref function. But you can use this !Ref function to reference to other resource in the Resource itself. Like EIP resource to the EC2 with !Ref

# RESOURCES

- Resources are the core of the CloudFormation templates (Mandatory)
- They represent the different AWS components that will be created and configured
- Resources are declared and can reference each other.
- AWS figures out creation, updates and deletes of resources for us.
- There are over 224 types of resources (!)
- Resources types identifiers are of the form:

**AWS::aws-product-name::data-type-name**

To look for the resource you can always google it and get it from there.


For example in the Resource block we give the small indentation and we defined all the resources under the Resource block. Line7.

**Type** and **Properties** has to be defined.

# MAPPING

- Mapping are fixed variables within your CloudFormation Templates
- They are handy to differentiate between different environments (dev vs prod), regions (AWS regions), AMI types etc.
- All the values are hardcoded within the template

Example:

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

```
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"
```

## When would you use mapping vs parameters?

- Mapping are great when you know in advance all the values that can be taken and that they can be deducted from variables such as:
  - ☑ Region
  - ☑ Availability Zone
  - ☑ AWS Account
  - ☑ Environment (dev vs prod)
- 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]**

## Concept: Pseudo Parameters

- AWS offers us pseudo parameters in any CloudFormation template.
- These can be used at any time and are enabled by default

| Reference Value | Example Return Value |
|---|---|
| AWS::AccountId | 1234567890 |
| AWS::NotificationARNs | [arn:aws:sns:us-east-1:123456789012:MyTopic] |
| AWS::NoValue | Does not return a value. |
| AWS::Region | us-east-2 |
| AWS::StackId | arn:aws:cloudformation:us-east-1:123456789012:stack/MyStack/1c2fa620-982a-11e3-aff7-50e2416294e0 |
| AWS::StackName | MyStack |

```
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
```

# OUTPUTS

- ☑ The Outputs section declares Optional outputs values that we can import into other stacks (if you export then first)!
- ☑ You can also view the outputs in the AWS Console or in using AWS CLI
- ☑ They are very useful example if you define a network CloudFormation and outputs the variables such as VPC ID and SubnetIDs
- ☑ It is the best way to perform some collaboration cross stack, as you let expert handle their own part of the stack
- ☑ You can not delete a CloudFormation Stack if it is outputs are being referenced by another CloudFormation stack.

**Output Example:**
Create a SSH Security Group as a part of one template
We create an output that references that security group.

We have the output and we have the name of the Security Group. Then we give a description and **Value:**we are basically referencing it to the Security Group that is in the Resource and **Export:** you can use it to export value name that will be used to reference it in another resource.

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

**Cross Stack Reference**

- For this we use the **Fn::ImportValue** function.
- You can not 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
```

In the security group: you can use **ImportValue** to reuse the Security Group.

# CloudFormation Conditions

- 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:
  - ☑ Environments
  - ☑ AWS Region
  - ☑ Any parameter value
- Each condition can reference another condition, parameter value or mapping.

**How to define a condition?**

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

- The logical ID is for you to choose.It is 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**

- Condition can be applied to resource / outputs / etc..

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

## CloudFormation Must Know Intrinsic Functions

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

## Fn::Ref

- The **Fn::Ref** function can be leverage to reference
  - ☑ Parameters => returns the value of the parameters
  - ☑ Resource => 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 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::Sub

Fn::Sub, or !Sub as a shorthand, is used to substitute variables from a text. It is a very handy function that will allow you fully customize your templates.

For Example, you can combine Fn::Sub with Reference or AWS Pseudo variables!
String must contain ${VariableName} and will substitute them

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

```
!Sub String
```

# User Data in EC2 for CloudFormation

- ☑ We can have user data at EC2 instance launch through the console
- ☑ We can also include it in CloudFormation
- ☑ The important thing to pass is the entire script through the function **Fn::Base64**
- ☑ Good to know: user data script log is in **/var/log/cloud-init-output.log**

3-user-data.yaml

```yaml
---
Parameters:
  SSHKey:
    Type: AWS::EC2::KeyPair::KeyName
    Description: Name of an existing EC2 KeyPair to enable SSH access to the instance

Resources:
  MyInstance:
    Type: AWS::EC2::Instance
    Properties:
      AvailabilityZone: us-east-1a
      ImageId: ami-009d6802948d06e52
      InstanceType: t2.micro
      KeyName: !Ref SSHKey
      SecurityGroups:
        - !Ref SSHSecurityGroup
      # we install our web server with user data
      UserData:
        Fn::Base64: |
          #!/bin/bash -xe
          yum update -y
          yum install -y httpd
          systemctl start httpd
          systemctl enable httpd
          echo "Hello World from user data" > /var/www/html/index.html

  # our EC2 security group
  SSHSecurityGroup:
    Type: AWS::EC2::SecurityGroup
    Properties:
      GroupDescription: SSH and HTTP
      SecurityGroupIngress:
      - CidrIp: 0.0.0.0/0
        FromPort: 22
        IpProtocol: tcp
        ToPort: 22
      - CidrIp: 0.0.0.0/0
        FromPort: 80
        IpProtocol: tcp
        ToPort: 80
```

**Fn::Base64** is the script where you have defined the user data and you need to have that "|" pipe in order to make your script work on the line 19, otherwise your script won't work.Our user data installs the simple apache script. And as we did before we need to upload the file and as we did before.

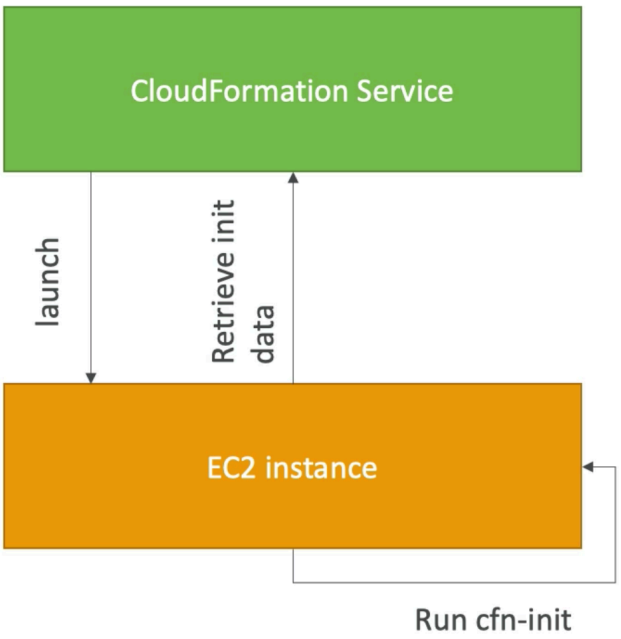
For the SSH key pair you can specify it in the console or if you don't have one, you can create it and put during he creation of the file.

```
~/aws-course ssh -i CloudFormationKeyPair.pem ec2-user@18.212.193.181
Warning: Permanently added '18.212.193.181' (ECDSA) to the list of known hosts.

       __|  __|_  )
       _|  (     /   Amazon Linux 2 AMI
      ___|\___|___|

https://aws.amazon.com/amazon-linux-2/
[ec2-user@ip-172-31-92-82 ~]$ cat /var/log/cloud-init-output.log
```

**This is where file for the user data will go good for the debug.**

# cfn-init

- AWS::CoudFormation::Init must be in the Metadata of resource
- With the **cfn-init** script, it helps make complex EC2 configuration service to get init data
- Logs go to **/var/log/cfn-init.log**

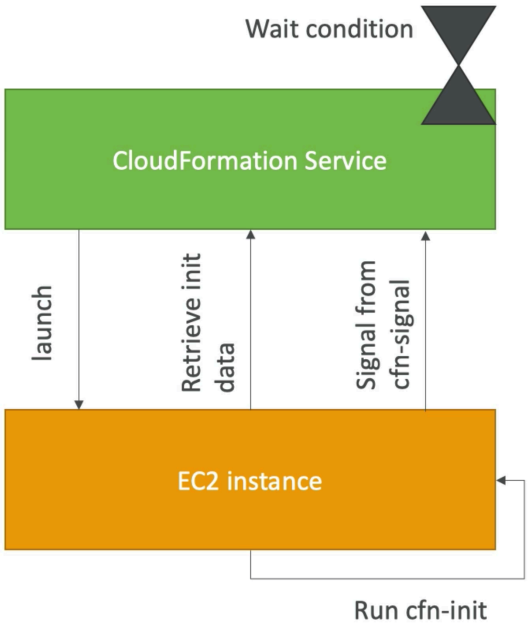



```yaml
---
Parameters:
  SSHKey:
    Type: AWS::EC2::KeyPair::KeyName
    Description: Name of an existing EC2 KeyPair to enable SSH access to the instance

Resources:
  MyInstance:
    Type: AWS::EC2::Instance
    Properties:
      AvailabilityZone: us-east-1a
      ImageId: ami-009d6802948d06e52
      InstanceType: t2.micro
      KeyName: !Ref SSHKey
      SecurityGroups:
        - !Ref SSHSecurityGroup
      # we install our web server with user data
      UserData:
        Fn::Base64:
          !Sub |
            #!/bin/bash -xe
            # Get the latest CloudFormation package
            yum update -y aws-cfn-bootstrap
            # Start cfn-init
            /opt/aws/bin/cfn-init -s ${AWS::StackId} -r MyInstance --region ${AWS::Region} ||
            error_exit 'Failed to run cfn-init'
      Metadata:
        Comment: Install a simple Apache HTTP page
        AWS::CloudFormation::Init:
          config:
            packages:
              yum:
                httpd: []
            files:
              "/var/www/html/index.html":
                content: |
                  <h1>Hello World from EC2 instance!</h1>
                  <p>This was created using cfn-init</p>
                mode: '000644'
            commands:
              hello:
                command: "echo 'hello world'"
            services:
              sysvinit:
                httpd:
                  enabled: 'true'
                  ensureRunning: 'true'
```

Ln18: Still a user data
Ln19: Still a Base64
Ln20: We are passing !Sub function.
Ln23: we are updating the **aws-cfn-bootstrap** script.
Ln25:
Ln26: Metadata starts
Ln27: installs apache comment
Ln28: reps aws cf init block.
Ln29: config starts, where you start defining entire configuration.
Ln30: starts with **packages**. Install httpd.
Ln33: **Files** you can define the content inside.
Ln39: You can define the commands.
Ln42: You can say what **services** it should start.and the rest is exact same thing.

# Cfn-signal & wait conditions

- We still do not know how to tell CloudFormation that the EC2 instance got properly configured after a **cfn-init**
- For this we can use the **cfn-signal** script!
  - ☑ We run cfn-signal right after cfn-init
  - ☑ Tell CloudFormation service to keep on going or fail
- We need to define **WaitCondtion:**
  - ☑ Block the template until it receives a signal from cfn-signal
  - ☑ We attach a CreationPolicy (also works on EC2, ASG)




```
7   Resources:
8     MyInstance:
9       Type: AWS::EC2::Instance
10      Properties:
11        AvailabilityZone: us-east-1a
12        ImageId: ami-009d6802948d06e52
13        InstanceType: t2.micro
14        KeyName: !Ref SSHKey
15        SecurityGroups:
16          - !Ref SSHSecurityGroup
17        # we install our web server with user data
18        UserData:
19          Fn::Base64:
20            !Sub |
21              #!/bin/bash -xe
22              # Get the latest CloudFormation package
23              yum update -y aws-cfn-bootstrap
24              # Start cfn-init
25              /opt/aws/bin/cfn-init -s ${AWS::StackId} -r MyInstance --region ${AWS::Region}
26              # Start cfn-signal to the wait condition
27              /opt/aws/bin/cfn-signal -e $? --stack ${AWS::StackId} --resource SampleWaitCondition
                --region ${AWS::Region}
28      Metadata:
29        Comment: Install a simple Apache HTTP page
30        AWS::CloudFormation::Init:
31          config:
32            packages:
33              yum:
34                httpd: []
35            files:
36              "/var/www/html/index.html":
37                content: |
38                  <h1>Hello World from EC2 instance!</h1>
39                  <p>This was created using cfn-init</p>
40                mode: '000644'
41            commands:
42              hello:
43                command: "echo 'hello world'"
44            services:
45              sysvinit:
46                httpd:
47                  enabled: 'true'
48                  ensureRunning: 'true'
49
50    SampleWaitCondition:
51      CreationPolicy:
52        ResourceSignal:
53          Timeout: PT2M
54      Type: AWS::CloudFormation::WaitCondition
55
56    # our EC2 security group
57    SSHSecurityGroup:
58      Type: AWS::EC2::SecurityGroup
59      Properties:
```

We have exit same parameters and exact same user data but on line 27, we have cfn init command to define the conditions and basically it is saying if the line25 will work execute the instance. And on line 27 we are going to signal SampleWaitCondition that is on line50 where we are passing some configuration to the SimpleWaitCondition like how long it should wait.

Line 53: Timeout:PT2M

# CF cfn-signal failures troubleshooting

- Ensure that the AMI you are using has the AWS CF helpers scripts installed. If the AMI doesn't include the helper scripts, you can also download them to your instance.
- Verify that the **cfn-init & cfn-signal** command was successfully run on the instances. You can view logs, such as **/var/log/cloud-init.log or /var/log/cfn-init.log** to help you to debug the instance launch.

```
28    Metadata:
29      Comment: Install a simple Apache HTTP page
30      AWS::CloudFormation::Init:
31        config:
32          packages:
33            yum:
34              httpd: []
35          files:
36            "/var/www/html/index.html":
37              content: |
38                <h1>Hello World from EC2 instance!</h1>
39                <p>This was created using cfn-init</p>
40              mode: '000644'
41          commands:
42            hello:
43              command: "echo 'boom' && exit 1"
44          services:
45            sysvinit:
46              httpd:
47                enabled: 'true'
48                ensureRunning: 'true'
```

•You can retrieve the logs by logging in to your instance, but you must disable rollback on failure or else AWS CF deletes the instance after your stack fails to create.

•Verify that the instance has a connection to the internet.If the instance is in VPC the instance should be able to connect to the internet through a NAT device if it is in a private subnet or through an Internet gateway if it is in a public subnet.

Same Script and we will just exit in the command line43 with the exit1 message, that says if the script is not working it will exit with error message.

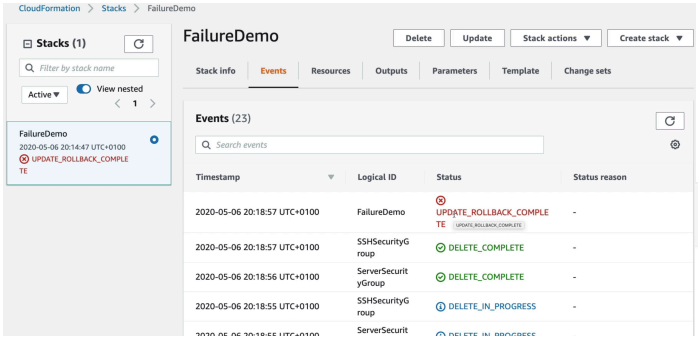
# CF Rollbacks

**Stack Creation Fails:**

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

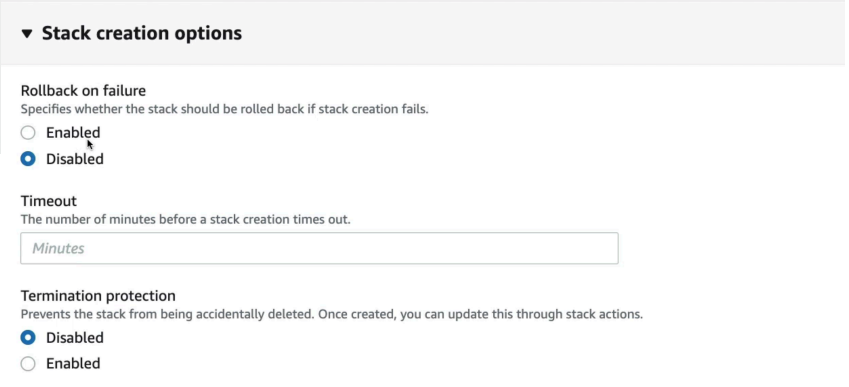
**Stack Update Fails:**

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




In case if the stack will fail with new update it will roll back to the previous condition. It will not place an update for us. That is what the roll back does for us.

But if we disable the rollback it will not go back to the previous condition but it will fail. Sometimes it is good for the troubleshooting. Nothing will be rolled back.



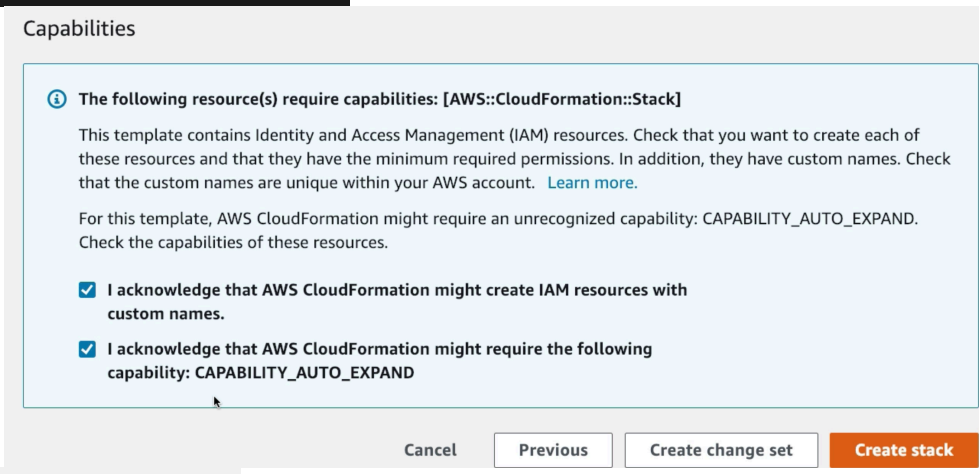
# CF Nested Stacks

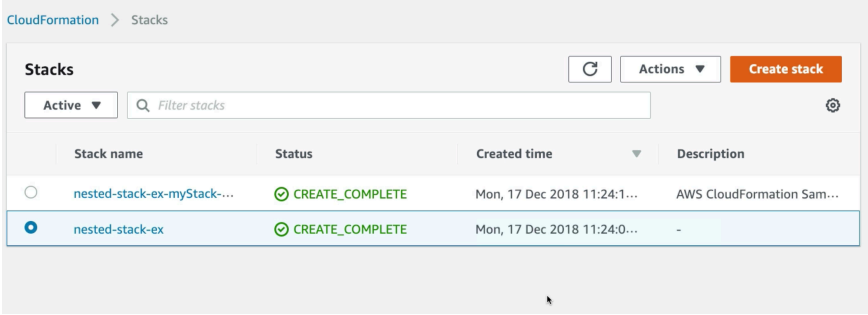
- Nested stacks are the stacks as part of other stacks
- They allow you to install isolate repeated patterns / common components in separate stacks and call them from other stacks.
- Example:
  - Load Balancer configuration that is re-used
  - Security Group that is re-used
- Nested Stacks are considered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

```yaml
Parameters:
  SSHKey:
    Type: AWS::EC2::KeyPair::KeyName
    Description: Name of an existing EC2 KeyPair to enable SSH access to the instance

Resources:
  myStack:
    Type: AWS::CloudFormation::Stack
    Properties:
      TemplateURL:
      https://s3.amazonaws.com/cloudformation-templates-us-east-1/LAMP_Single_Instance.template
      Parameters:
        KeyName: !Ref SSHKey
        DBName: "mydb"
        DBUser: "user"
        DBPassword: "pass"
        DBRootPassword: "passroot"
        InstanceType: t2.micro
        SSHLocation: "0.0.0.0/0"

Outputs:
  StackRef:
    Value: !Ref myStack
  OutputFromNestedStack:
    Value: !GetAtt myStack.Outputs.WebsiteURL
```

Nested stacks are basically the file that you can define all the configuration with the resource and upload it with using the stack. For Example: in our case our file that is in S3 bucket has the configuration of files that creates and EC2, DB and other resources.

This is the warning that tells you that you need to have all the proper IAAM rolls to securely deploy the stack.

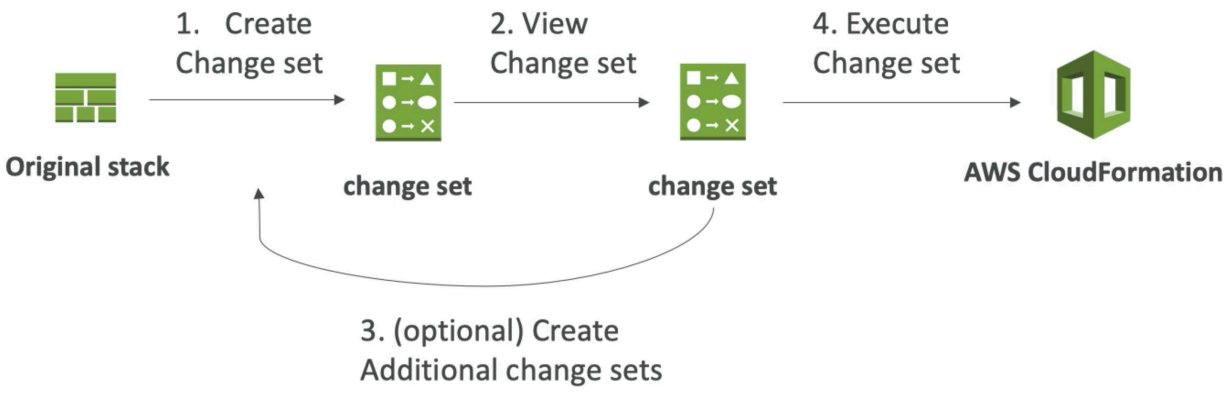


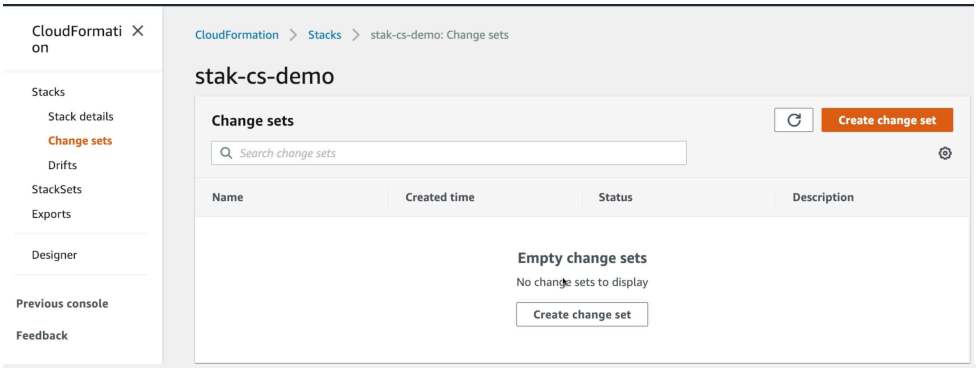

Warning: never update an child stack, what you usually do is you have to update the parent stack or you can delete the parent stack but never delete the child stack Or you have to delete all the provisioned resources manually

# ChangesSets

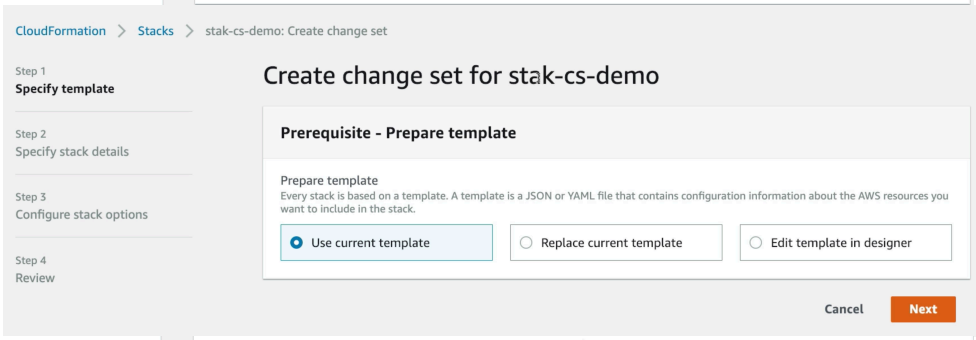
- When you update a stack, you need to know what changes before it happens for greater confidence.
- ChangeSets won't say if the update will be successful.



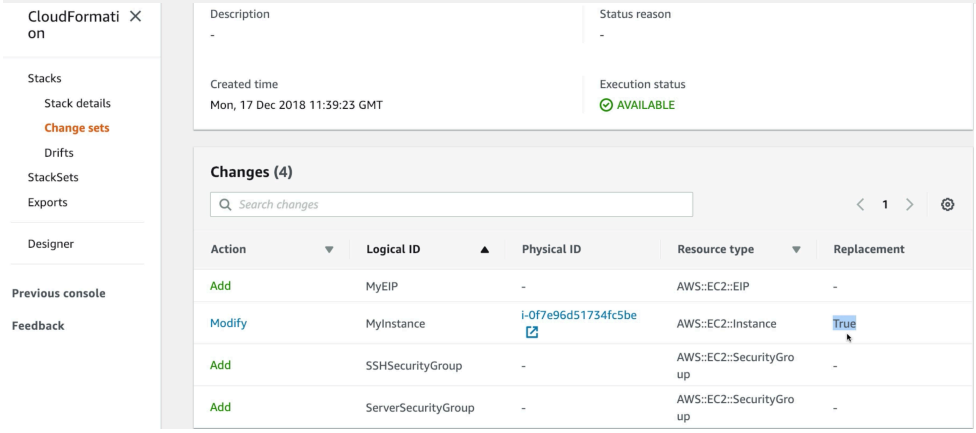

We have been using the changeSet so far until now. But to see the changeSet you need to upload the updated template file for that. And it will show us the changes that has been made. And this is how our change set looks like. But as you can see as of now we don't have any changeSet lets create it. So there are two ways to create it and one is you can right click on the existing stack or you can press create change stack.




We will go and replace the existing stack with the new one. And press create a change set. But our changeSet is not implemented yet. It will create a change but to implement the changes that we had defined in the changeSet we need to press execute.




And we can see the all the changes that will be implemented to our new infrastructure.



# Retain Data on Deletes

You can put DeletePolicy on any resource to control what happens when the CF template is deleted.

**DelationPolicy=Retain:**

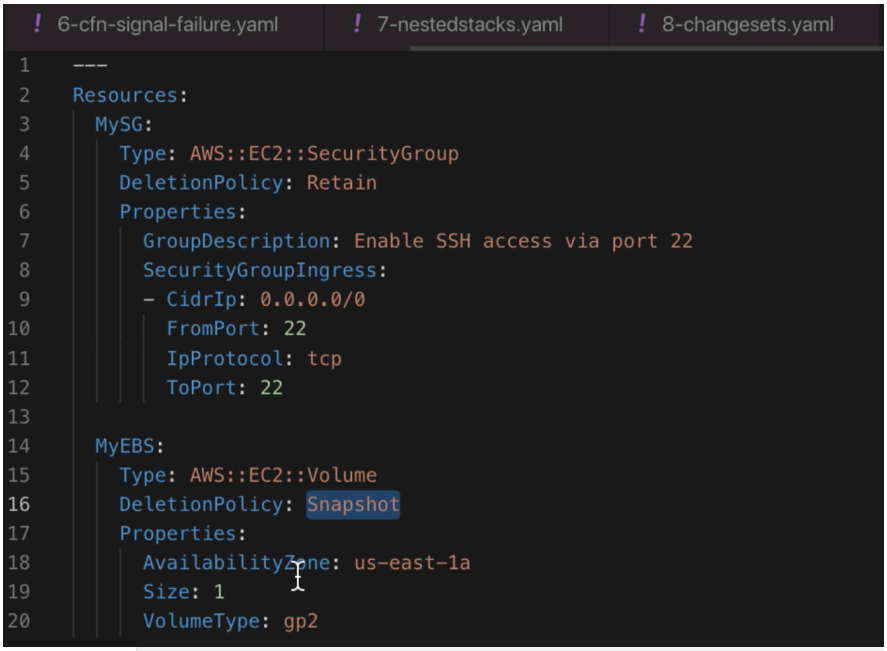
- Specify on resource to preserve / backup in case of CF deletes.
- To keep a resource, specify Retain (works for any resource / nested stack)

**DeletionPolicy=Snapshot:**

- EBS Volume, ElasticCache Cluster, ElasticCache Replication Group
- RDS DBInstace, RDS DBCluster, Redshift Cluster

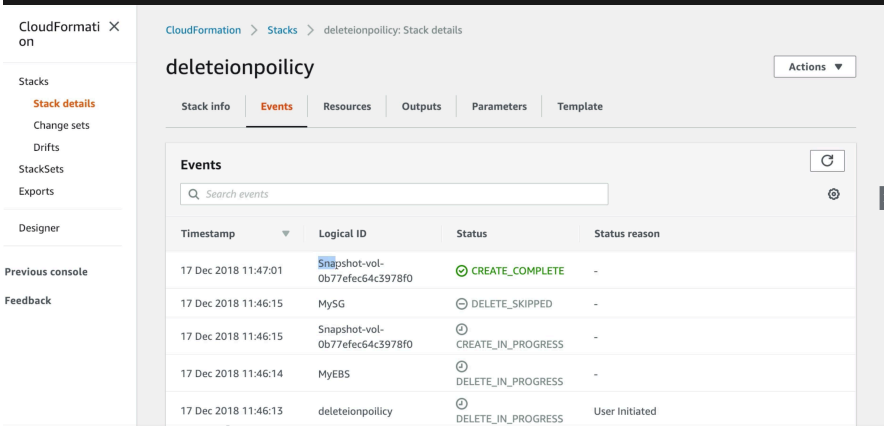
**DeletePolicy=Delete (default behavior):**

- Note:AWS::RDS::DBCluster resources, the default policy is Snapshot.
- Note: to delete an S3Bucket, you need to first empty the bucket of its content.




As you can see on the line5 we have enabled the deletionPolicy to Retain: It should keep our Security Group.

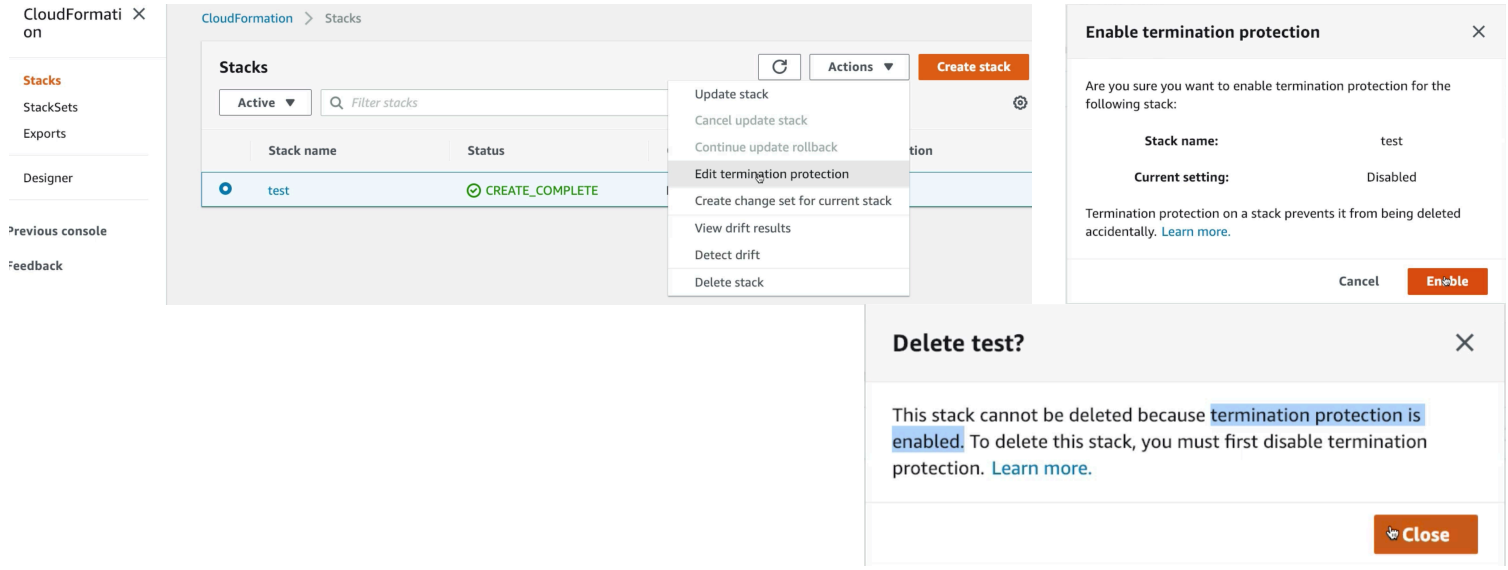
And in the second resource line 16, we have enabled the snapshot of our EBS volume. It should take a snapshot of it.




So what we need to remember is we have a snapshot of EBS volume and Security Group existing. Since we specified it in the resource section.

# Termination Protection on Stacks

To prevent accidental deletes of the CF templates use **TerminationProtection**.




## We will create an AutoScaling Group

auto-scaling > cloudformation > ! 0-asg-creation-policy.yml > { } Parameters > { } LatestLinuxAmiId >

```yaml
Parameters:
  LatestLinuxAmiId:
    Type: 'AWS::SSM::Parameter::Value<AWS::EC2::Image::Id>'
    Default: '/aws/service/ami-amazon-linux-latest/amzn2-ami-hvm-x86_64-gp2'

Resources:
  AutoScalingGroup:
    Type: AWS::AutoScaling::AutoScalingGroup
    Properties:
      AvailabilityZones:
        Fn::GetAZs:
          Ref: "AWS::Region"
      LaunchConfigurationName:
        Ref: LaunchConfig
      DesiredCapacity: '3'
      MinSize: '1'
      MaxSize: '4'
    CreationPolicy:
      ResourceSignal:
        Count: '3'
        Timeout: PT15M

  LaunchConfig:
    Type: AWS::AutoScaling::LaunchConfiguration
    Properties:
      ImageId: !Ref LatestLinuxAmiId
      InstanceType: t2.micro
      UserData:
        "Fn::Base64":
          !Sub |
            #!/bin/bash -xe
            yum update -y aws-cfn-bootstrap
            /opt/aws/bin/cfn-signal -e $? --stack ${AWS::StackName} --resource
            AutoScalingGroup --region ${AWS::Region}
```

**CerationPolicy** The creation policy requires three success signals and times out after 15 minutes. To have instances wait for an Elastic Load Balancing health check before they signal success, add a health-check verification by using the cfn-init helper script.