Building a VPC with CloudFormation - Part 2

Leia em portuguÃªs

Key Takeaways

- For a modest amount of effort, a CloudFormation template can be made flexible and powerful by using parameters and conditions.
- Mappings can be used to conditionally select values from a lookup table
- Outputs can be used to clearly designate specific stack resources to other stacks.
- Teams can use exported outputs as inter-team communications

In part one of this article, we looked at how to use Infrastructure as Code, and CloudFormation in particular, to create and maintain an AWS VPC. The CloudFormation template we created provides a simple, reusable artifact we can use any time we need to create a simple VPC.

However, this template is not as flexible as it can be. We would like to have a template that can build a VPC with a varying number of subnets to handle development vs test vs production usage. We would like something that could create public-only subnets if we needed to quickly create something for demo / POC purposes. Or we might wish to use a NAT instance instead of a NAT gateway.

Instead of creating separate templates for these cases, we can make our existing template more flexible by using Parameters, Conditions, Mappings, and Outputs. Since this is the second article in the series, you should probably acquaint yourself with the original article and template. I'll deliver this narrative from the perspective of taking the original template and enhancing it.

Cut to the Chase: The source code CloudFormation template described by this article is found here on GitHub. Feel free to download, modify, and use this template however you like (though I will not accept liability for misuse).

Variable Number of Subnets / AZs

Availability Zones: AWS has made it easy and inexpensive to take advantage of multiple Availability Zones (AZs) within a given region. For an overly simplistic explanation, you can think of an Availability Zone as a huge independent datacenter. AZs within a region are connected to each other by high-speed, low-latency, privately operated links. They are close enough to each other to support synchronous communications, but far enough apart to mitigate the effect of natural disasters, power outages, etc. Exactly how far apart is not disclosed, and not really relevant.

Two AZs is a good number to achieve basic high-availability at minimum cost. But sometimes a single AZ is better for simple cases like demos or POC's. Other times three is desired for marginally improved high availability, or to make better use of the spot market. So let's adjust the template to make the number of AZ's variable.

Using the template from article 1, add the following section above the "Resources" section.

```
Parameters:
  NumberOfAZs:
    Type: Number
    AllowedValues:
    - 1
    - 2
    - 3
    Default: 2
    Description:  How many Availability Zones do you wish to utilize?
```

YAML Basics: In YAML, indentation with double spaces indicates hierarchy (no tabs!). The dash character "-" is YAML syntax for defining a 'sequence', multiple values belonging to the same group. The parameters section is commonly placed above the resources section, but technically it can reside anywhere in the template.

NumberOfAZs: This entry defines an input parameter for the template. When creating a stack using this template within the AWS Management Console, the UI will prompt you to enter "NumberOfAZs" with the given description near the field. Since we've provided "AllowedValues", the input field will be a drop-down box with the choices of 1, 2, and 3. The value 2 will be used if we make no other selection. The valid types of parameters is defined here, we could have used Number or String in this case.

One goal we should pursue is the ability to use this single template to create a stack in any region we like. At the time of this writing, most regions now have a minimum of three availability zones, but some do not (Montreal, Mumbai, Beijing, Seoul only have two). Selecting three AZs in these regions will result in an error. Whether it is worthwhile to limit the template's flexibility to eliminate an awkward error in the minority case is for you to decide.

CLI usage: When creating a stack via the AWS command line interface (CLI), input parameters still have purpose. We can supply a value for this parameter if we like, or simply take the default value. Providing an input value outside the allowed set will result in an error.

Now that we have a way to specify the desired number of AZ's, we need to alter the rest of the template to have CloudFormation build the subnets to match what we want.

Conditions Section

In order to have CloudFormation build one, two, or three subnets, we will define some "Conditions" that can be used in the resources section. Add this code below the Parameters section and above the Resources section:

```
Conditions:
  BuildPublicB:         !Not [ !Equals [ !Ref NumberOfAZs, 1 ]]
  BuildPublicC:         !Equals [ !Ref NumberOfAZs, 3 ]
```

Conditions are boolean (true/false) expressions that we will use elsewhere in the template. Here we are creating two, one indicating if we want to build the "B" subnets, one indicating if we want to build the "C" subnets. Since "1" is the minimum number allowed by NumberOfAZs, we will always build the "A" subnets.

BuildPublicB: This expression is checking if the selected NumberOfAZs is anything other than 1. Since we don't have greater than, less than intrinsic functions in CloudFormation, we will use the !Equals function to reference the input parameter and check its equality to "1". The !Not is negating that result (false becomes true, true becomes false). The resulting boolean is stored in BuildPublicB were it can be referred to elsewhere in the template.

BuildPublicC: This expression is simpler, the NumberOfAZs is either "3" (our maximum allowed) or not. We will only build our PublicSubnetC if this is true.

Now that we have clearly defined conditions about which subnets to create, we can use these to affect the creation of the actual resources.

The Condition Attribute

In the original template from article 1, we created PublicSubnetB with this code:

```
  PublicSubnetB:
    Type: AWS::EC2::Subnet
    Properties:
      VpcId: !Ref VPC
      CidrBlock: 10.1.20.0/24
      AvailabilityZone: !Select [ 1, !GetAZs ]    # Get the second AZ in the list
      Tags:
      - Key: Name
        Value: !Sub ${AWS::StackName}-Public-B
```

Now notice the replacement code below, specifically the new “Condition” attribute:

```
PublicSubnetB:
    Type: AWS::EC2::Subnet
    Condition: BuildPublicB
    Properties:
      VpcId: !Ref VPC
      CidrBlock: 10.1.20.0/24
      AvailabilityZone: !Select [ 1, !GetAZs ]    # Get the second AZ in the list
      Tags:
      - Key: Name
        Value: !Sub ${AWS::StackName}-Public-B
```

The condition attribute is an available option on any CloudFormation resource. Essentially it is saying, “only create this resource if the BuildPublicB condition is true”. When it is false, the entire resource is bypassed during stack creation - there will be no PublicSubnetB.

While we are here, let’s add a third public subnet, but only if the BuildPublicC condition allows it:

```
  PublicSubnetC:
    Type: AWS::EC2::Subnet
    Condition: BuildPublicC
    Properties:
      VpcId: !Ref VPC
      CidrBlock: 10.1.30.0/24
      AvailabilityZone: !Select [ 2, !GetAZs ]    # Get the third AZ in the list
      Tags:
      - Key: Name
        Value: !Sub ${AWS::StackName}-Public-C
```

You may wonder if there is a way to express the condition in-line on the resource rather than using the separate “Conditions” section. At the time of this writing there is not. But after writing many templates, I’ve come to appreciate the simple decoupling of the logical expression calculation from its usage. After all, these templates can become quite complex with in-line expressions, such as the ones you see here for AvailabilityZone or the Tag/Value.

Last step, the subnet to route table associations must be adjusted for the varying number of public subnets. Note the use of the condition attributes in the following:

```
  PublicSubnetBRouteTableAssociation:
    Type: AWS::EC2::SubnetRouteTableAssociation
    Condition: BuildPublicB
    Properties:
      SubnetId: !Ref PublicSubnetB
      RouteTableId: !Ref PublicRouteTable
  PublicSubnetCRouteTableAssociation:
    Type: AWS::EC2::SubnetRouteTableAssociation
    Condition: BuildPublicC
    Properties:
      SubnetId: !Ref PublicSubnetC
      RouteTableId: !Ref PublicRouteTable
```

Not shown is the association for PublicSubnetA. Since it is always present in the stack it requires no conditional attribute. Likewise, the PublicRouteTable itself is always needed. At this point, our stack will create one, two, or three public subnets based on our input parameter. Let’s look at the private subnets...

Private Subnets.

Let's imagine we'd like to use the VPC produced by this template for some quick public-facing demos. Having private subnets or a NAT in such a VPC would be overkill, and would take longer to create, so let's add a parameter to allow us to designate public-only subnets. In the "Parameters" section add:

```
  PrivateSubnets:
    Type: String
    AllowedValues:
    - True
    - False
    Default: True
    Description: Do you want to create private subnets in addition to public subnets
```

Here we are defining an input parameter to control whether any private subnets are created. I wish CloudFormation provided a "Boolean" input type for yes/no cases like this, but we'll have to make do with a String that accepts only "True" or "False".

Let's add these conditions inside the Conditions section to act on the input value; this is going to get a bit complicated:

```
  BuildPrivateSubnets: !Equals [ !Ref PrivateSubnets, True ]
  BuildPrivateA:       !Equals [ !Ref PrivateSubnets, True ]
  BuildPrivateB:       !And[!Not[!Equals[!Ref NumberOfAZs,1]],!Equals[!Ref PrivateSu
  BuildPrivateC:       !And[!Equals[!Ref NumberOfAZs,3],!Equals[!Ref PrivateSubnets,
```

BuildPrivateSubnets: This is a clear, succinct condition that directly expresses the input parameter. There are a few spots where we will build something based on whether there are any private subnets at all (i.e. the NAT)

BuildPrivateA: A synonym for "BuildPrivateSubnets". Not strictly necessary, but you'll see how clean the resulting code is when we are done. It's a bit of a shame that we can't reference a condition from another condition, that would allow a nice way to break down complex logic.

BuildPrivateB: The logic here is saying “Only build PrivateSubnetB if 1) we want to use more than one AZ, and if 2) we want to build private subnets”

BuildPrivateC: The logic here: “Only build PrivateSubnetC if 1) we want to utilize three AZ’s, and if 2) we want to build private subnets”.

Now we can convert the private subnet definitions from our original template to use condition attributes, like this:

```
  PrivateSubnetA:
    Type: AWS::EC2::Subnet
    Condition: BuildPrivateA
    Properties:
      VpcId: !Ref VPC
      CidrBlock: 10.1.50.0/24
      AvailabilityZone: !Select [ 0, !GetAZs ]    # Get the first AZ in the list
      Tags:
      - Key: Name
        Value: !Sub ${AWS::StackName}-Private-A
  PrivateSubnetB:
    Type: AWS::EC2::Subnet
    Condition: BuildPrivateB
    Properties:
      VpcId: !Ref VPC
      CidrBlock: 10.1.60.0/24
      AvailabilityZone: !Select [ 1, !GetAZs ]    # Get the second AZ in the list
      Tags:
      - Key: Name
        Value: !Sub ${AWS::StackName}-Private-B
  PrivateSubnetC:
    Type: AWS::EC2::Subnet
    Condition: BuildPrivateC
    Properties:
      VpcId: !Ref VPC
      CidrBlock: 10.1.70.0/24
      AvailabilityZone: !Select [ 2, !GetAZs ]    # Get the third AZ in the list
      Tags:
      - Key: Name
        Value: !Sub ${AWS::StackName}-Private-C
```

Again, the only modification from our starting template is the addition of the Condition attributes. Plus, we have added PrivateSubnetC, which was relatively easy to clone from the definitions of PrivateSubnetA and PrivateSubnetB.

The subnet route table associations will require modification. There is no need for a subnet association if there is no subnet:

```
  PrivateSubnetARouteTableAssociation:
    Type: AWS::EC2::SubnetRouteTableAssociation
    Condition: BuildPrivateA
    Properties:
      SubnetId: !Ref PrivateSubnetA
      RouteTableId: !Ref PrivateRouteTable
  PrivateSubnetBRouteTableAssociation:
    Type: AWS::EC2::SubnetRouteTableAssociation
    Condition: BuildPrivateB
    Properties:
      SubnetId: !Ref PrivateSubnetB
      RouteTableId: !Ref PrivateRouteTable
  PrivateSubnetCRouteTableAssociation:
    Type: AWS::EC2::SubnetRouteTableAssociation
    Condition: BuildPrivateC
    Properties:
      SubnetId: !Ref PrivateSubnetC
      RouteTableId: !Ref PrivateRouteTable
```

The NAT Gateway.

Since our template now conditionally creates the private subnets, we need to adjust our NAT Gateway and RouteTable entries accordingly. First, the NAT Gateway; there is no longer a reason to create it or its associated Elastic IP Address if we do not wish to build private subnets:

```
  # A NAT Gateway will be built and used if the user selected Private subnets and a C
  NATGateway:
    Type: AWS::EC2::NatGateway
    Condition: BuildPrivateSubnets
    Properties:
      AllocationId: !GetAtt ElasticIPAddress.AllocationId
      SubnetId: !Ref PublicSubnetA
      Tags:
      - Key: Name
        Value: !Sub NAT-${AWS::StackName}
  ElasticIPAddress:
    Type: AWS::EC2::EIP
    Condition: BuildPrivateSubnets
    Properties:
      Domain: VPC
```

The only change from our original template is the Condition attributes; we only wish to build these if we have chosen to build private subnets.

Next, the condition may indicate we won't need the private route table or its entry:

```
  # Here is a private route table:
  PrivateRouteTable:
    Type: AWS::EC2::RouteTable
    Condition: BuildPrivateSubnets
    Properties:
      VpcId: !Ref VPC
      Tags:
      - Key: Name
        Value: Private
  PrivateRoute1:            # Private route table can access web via NAT (created bel
    Type: AWS::EC2::Route
    Condition: BuildPrivateSubnets
    Properties:
      RouteTableId: !Ref PrivateRouteTable
      DestinationCidrBlock: 0.0.0.0/0
```

```
      # Route traffic through the NAT Gateway:
      NatGatewayId: !Ref NATGateway
```

At this point, our stack will bypass creation of any private subnets, route tables, or NATs when "BuildPrivateSubnets" is false. Our template has the ability to create between one and six subnets based on parameter input. Pretty flexible, and really not that much work to achieve this.

Optional: NAT Type

For further flexibility, let's have our template provide an alternative to the NAT Gateway. The built-in managed service is great for production use, but we might find it a bit expensive for POC's. In the days before NAT Gateways, we used regular EC2 instance configured to provide NAT support, and there are advantages and disadvantages either way. So in the name of experimentation, let's add a parameter to allow this choice:

```
  NATType:
    Type: String
    AllowedValues:
    - "EC2 NAT Instance"
    - "NAT Gateway"
    Default:  "NAT Gateway"
    Description:  What type of NAT to use for private instances to communicate with
```

…and some additions to the Conditions section:

```
  BuildNATGateway:  !And[!Equals[!Ref PrivateSubnets,True],!Equals[!Ref NATType, "NA
  BuildNATInstance: !And[!Equals[!Ref PrivateSubnets,True],!Equals[!Ref NATType, "EC
```

The first statement is essentially, "Build a NAT Gateway if 1) we have chosen to build private subnets, and if 2) we have chosen to build a NAT Gateway". The second is "Build an EC2 instance to serve as a NAT if 1) we have chosen to build private subnets, and if 2) we have opted to use an EC2 instance for NAT".

Our NAT Gateway / Elastic IP Address conditions described earlier will require adjustment, we now want to control their creation based on the BuildNATGateway condition:

```
  # A NAT Gateway will be built and used if the user selected Private subnets and a C
  NATGateway:
    Type: AWS::EC2::NatGateway
    Condition: BuildNATGateway
    Properties:
      AllocationId: !GetAtt ElasticIPAddress.AllocationId
      SubnetId: !Ref PublicSubnetA
      Tags:
      - Key: Name
        Value: !Sub NAT-${AWS::StackName}
  ElasticIPAddress:
    Type: AWS::EC2::EIP
    Condition: BuildNATGateway
    Properties:
      Domain: VPC
```

The EC2 based NAT instance will require some new construction. First, our EC2 instance will require an AMI, but the value for AMI ID will vary by which region we are running in. To allow this single template to be used in any region, add the following Mappings section to your template before the Condition section (though technically sections can be located in any order, and some people like to put this one near the bottom):

```
 Mappings:
  #  This is the Amazon Linux 2 AMI.  Adjust these values as needed, they can change
  AmazonLinuxAMI:
    us-east-1:
      AMI: ami-04681a1dbd79675a5    # N Virginia
    us-east-2:
      AMI: ami-0cf31d971a3ca20d6    # Ohio
    us-west-1:
      AMI: ami-0782017a917e973e7    # N California
    us-west-2:
      AMI: ami-6cd6f714             # Oregon
    eu-west-1:
```

```
    AMI: ami-0bdb1d6c15a40392c    # Ireland
  eu-central-1:
    AMI: ami-0f5dbc86dd9cbf7a8    # Frankfurt
  sa-east-1:
    AMI: ami-0ad7b0031d41ed4b9    # Sao Paulo
  ap-southeast-1:
    AMI: ami-01da99628f381e50a    # Singapore
  ap-southeast-2:
    AMI: ami-00e17d1165b9dd3ec    # Sydney
  ap-northeast-1:
    AMI: ami-08847abae18baa040    # Tokyo
```

This mapping section defines the AMI ID value for the Amazon Linux 2 OS. The ID value varies according to the region the stack is being created in. In a bit we will see how this mapping table is used when we define the EC2 instance resource. But before moving on there are some important points to mention: 1) comments are your friend, 2) I didn't provide values for every region, 3) these values are current as of this article's publish date, from time to time the EC2 team will publish new, improved versions of the AMI which you should probably use.

It wasn't hard to find these values. I simply used the EC2 instance creation wizard in the AWS management console. Once I reached the AMI selection page, I used the region selection to pop around the globe collection values. I should also point out that there are more advanced techniques which supplant the need to have any mapping table (e.g. a Parameter Store lookup or CloudFormation custom resource backed by a Lambda function), but I did not want to complicate this article too much.

Next, our EC2 instance will need a Security Group:

```
# A security group for our NAT.  Ingress from the VPC IPs only.  Egress is TCP & U
NATSecurityGroup:
  Type: AWS::EC2::SecurityGroup
  Condition: BuildNATInstance
  DependsOn: AttachGateway
  Properties:
    GroupName: !Sub NATSecurityGroup-${AWS::StackName}
    GroupDescription: Enable internal access to the NAT device
    VpcId: !Ref VPC
```

```
      SecurityGroupIngress:
      - IpProtocol: tcp
        FromPort: '0'
        ToPort: '1024'
        CidrIp: !GetAtt VPC.CidrBlock
      SecurityGroupEgress:
      - IpProtocol: tcp
        FromPort: '0'
        ToPort: '65535'
        CidrIp: 0.0.0.0/0
      - IpProtocol: udp
        FromPort: '0'
        ToPort: '65535'
        CidrIp: 0.0.0.0/0
```

To keep this article brief I'm not going to explain all of the complexities here, but the summary is:

- We only create this if the condition for BuildNATInstance is true.
- We don't want to attempt to create this until after the Internet Gateway is attached to the VPC (see previous article)
- The Security Group's name will be based on the name of our CloudFormation stack (see previous article)
- The Security Group only allows inbound traffic from the VPC's own internal address range. Only private IPs within the VPC can send traffic to our NAT.
- Outbound traffic can be TCP or UDP based, and can go basically anywhere.
- See the AWS::EC2::SecurityGroup for full information on this resource.

Next, the EC2 instance:

```
  # A NAT Instance will be used if the user selected Private subnets and EC2-based N
  NATInstance:
    Type: AWS::EC2::Instance
    Condition: BuildNATInstance
    DependsOn: PublicRoute1                          # Must have route to IGW estab
    Properties:
      ImageId: !FindInMap [ AmazonLinuxAMI, !Ref "AWS::Region", AMI]  # lookup from
```

```
      InstanceType: t2.small                          # Any instance type is fine
      NetworkInterfaces:
      - DeviceIndex: '0'
        SubnetId: !Ref PublicSubnetA                  # Any public subnet is fine
        AssociatePublicIpAddress: true                # We will need a public IP add
        GroupSet: [!Ref NATSecurityGroup]             # Plug in the security group
      SourceDestCheck: false  # NATs don't work if EC2 matches source with destinati
      Tags:
      - Key: Name
        Value: !Sub NAT-${AWS::StackName}
      UserData:      #  This code is NAT code.  Last line signals completion:
        Fn::Base64: !Sub |
          #!/bin/bash
          yum -y update
          yum install -y aws-cfn-bootstrap
          echo 1 > /proc/sys/net/ipv4/ip_forward
          echo 0 > /proc/sys/net/ipv4/conf/eth0/send_redirects
          /sbin/iptables -t nat -A POSTROUTING -o eth0 -s 0.0.0.0/0 -j MASQUERADE
          /sbin/iptables-save > /etc/sysconfig/iptables
          mkdir -p /etc/sysctl.d/
          cat << NatConfFileMarker > /etc/sysctl.d/nat.conf
          net.ipv4.ip_forward = 1
          net.ipv4.conf.eth0.send_redirects = 0
          NatConfFileMarker
          /opt/aws/bin/cfn-signal -e 0 --resource NATInstance --stack ${AWS::StackNa
    # This NATInstance is only complete when you get 1 signal back within 5 minutes'
    CreationPolicy:
      ResourceSignal:
        Count: 1
        Timeout: PT5M
```

Briefly:

- Condition: We will only build this instance if BuildNATInstance is true.
- DependsOn: We won't attempt to build this until PublicRoute1 is completely built, i.e. we must be connected to the internet. This will be critical to allow the "yum" command in UserData run as expected (below)

- ImageID: The AMI to use comes from the mapping table established earlier. “AWS::Region” is a pseudo parameter which always tells us the region the stack is being created in. Essentially, we are asking CloudFormation to look up the region in the mapping table and use the resulting AMI.
- SubnetId: We are placing this EC2 NAT instance in a public subnet to facilitate the outbound communications. It is a bit simplistic to use a single NAT instance in a single public subnet like this, but the main intent of this article is to demonstrate basic flexibility, not exhaustive coverage of best practices. Feel free to adjust and improve.
- AssociatePublicIPAddress: The NAT needs a public IP address to converse with parties on the public internet.
- GroupSet: The NAT is associated with the security group defined earlier. This parameter requires a list of security groups, not a single value. So the square brackets “[ ]” are added to coerce the single value into a list structure.
- SourceDestCheck: This tells EC2 to bypass the normal checks it makes to ensure that the instance is either the source or destination of any traffic it receives, which is not the case for a NAT. The simple explanation is this is something we must do for NAT to work. See source / destination check for a more in-depth explanation.
- UserData: The Linux script you see here establishes the NAT capability. A full explanation of this script is beyond the scope of this article. The !Sub intrinsic function looks for ${} expressions and substitutes them. such as the ${AWS::StackName}. The final line of the script, cfn-signal is a CloudFormation specific function which signals the CloudFormation stack when the script is complete
- Fn::Base64: !Sub: A UserData script must be Base64 encoded, which is easy to do with the Fn::Base64 intrinsic function. Ordinarily I would use !Base64 shortcut syntax, but I also need to use the !Sub intrinsic function to replace placeholders in the script. Using the !Sub within !Base64 is valid Cloud Formation, but is not valid YAML, so we have to use the full function name for the outer function. Ugh.
- CreationPolicy: Ordinarily, CloudFormation considers a resource to be completely created when the underlying service says it is created. For EC2 instances, this is essentially when the operating system has begun starting up. However, in order for this EC2 NAT instance to be usable by anyone else in this stack, the User Data script needs to have finished. The CreationPolicy is essentially saying “this resource is not actually complete until we receive one signal (from the cfn-signal command) within 5 minutes”

Whew! that’s a lot! But the beauty of CloudFormation, or Infrastructure as Code in general, is that I only need to write this once. See AWS::EC2::Instance for full details on all of these settings.

Finally, we will need to adjust the PrivateRoute we built earlier. We will need to route outbound traffic to the NAT Gateway OR the NAT Instance, depending on which one was created:

```
PrivateRoute1:            # Private route table can access web via NAT (created belo
   Type: AWS::EC2::Route
   Condition: BuildPrivateSubnets
   Properties:
     RouteTableId: !Ref PrivateRouteTable
     DestinationCidrBlock: 0.0.0.0/0
     # If we are using a NAT Instance, route traffic through the NAT Instance:
     InstanceId:   !If [ BuildNATInstance, !Ref NATInstance, !Ref "AWS::NoValue" ]
     # Otherwise if we are using a NAT Gateway, route traffic through the NAT Gatewa
     NatGatewayId: !If [ BuildNATGateway, !Ref NATGateway, !Ref "AWS::NoValue" ]
```

Notice the InstanceId and NatGatewayId properties, according to the AWS::EC2::Route documentation they are mutually exclusive. The InstanceId is used in cases where we are routing traffic to an EC2 instance. The !If intrinsic function is only setting this value to the NATInstance if we have chosen to BuildNATInstance. The AWS::NoValue is more than it would appear, not only is it saying there is no value to set, but CloudFormation understands this to mean that there is no need to set this attribute at all. The mirror logic on NatGatewayId sets the value to NATGateway if we have chosen to BuildNATGateway. Since the conditions are mutually exclusive, only one of these is set, and our outbound traffic will use the NATInstance or the NATGateway, ultimately based on our original input decision.

Optional: Template Metadata

One other adjustment we can make on our revised template is cosmetic. We would like to control the input order of parameters more closely, prompting for the more essential choices first. To to this, add a Metadata section before the Parameters section (though some people like to put this at the bottom of the template):

```
Metadata:
 # Control the UI display when running this template from the AWS Management Console
 AWS::CloudFormation::Interface:
   ParameterGroups:
     - Label:
         default: "Network Configuration"
       Parameters:
         - NumberOfAZs
```

```
        - PrivateSubnets
        - NATType
```

Now when using this template to create a stack via the AWS Management Console, the parameters page will prompt the operator for “Network Configuration”, and present the parameters in the desired order. When using the CLI, this section has no impact.

Outputs

The template we have created is a nice, general purpose template that can be used as the starting point for other CloudFormation stacks that need a VPC. We want to make it easy to provide values from this stack as inputs into other stacks. This is especially relevant in most IT organizations where responsibilities are segmented between teams, and those permitted to manage network resources are not the same as those permitted to build resources which use the network. Providing output values from a stack can be done by creating an Outputs section:

```
Outputs:
 VPC:
   Description: VPC of the base network
   Value: !Ref VPC
   Export:
     Name: !Sub ${AWS::StackName}-VPC
 PublicSubnetA:
   Description: First Public Subnet
   Value: !Ref PublicSubnetA
   Export:
     Name: !Sub ${AWS::StackName}-PublicSubnetA
 PublicSubnetB:
   Description: Second Public Subnet
   Condition: BuildPublicB
   Value: !Ref PublicSubnetB
   Export:
     Name: !Sub ${AWS::StackName}-PublicSubnetB
 PublicSubnetC:
   Description: Third Public Subnet
   Condition: BuildPublicC
```

```
    Value: !Ref PublicSubnetC
    Export:
      Name: !Sub ${AWS::StackName}-PublicSubnetC
  PrivateSubnetA:
    Condition: BuildPrivateSubnets
    Description: First Private Subnet
    Value: !Ref PrivateSubnetA
    Export:
      Name: !Sub ${AWS::StackName}-PrivateSubnetA
  PrivateSubnetB:
    Condition: BuildPrivateB
    Description: Second Private Subnet
    Value: !Ref PrivateSubnetB
    Export:
      Name: !Sub ${AWS::StackName}-PrivateSubnetB
  PrivateSubnetC:
    Condition: BuildPrivateC
    Description: Third Private Subnet
    Value: !Ref PrivateSubnetC
    Export:
      Name: !Sub ${AWS::StackName}-PrivateSubnetC
```

At its most basic, these output entries display the relevant values on the AWS Management Console / CLI JSON output upon completion of the stack. Notice the inclusion of Conditional attributes to only emit values for resources which we actually created.

The part to notice is the Export/Name. This is producing a region-wide name by which this resource can be referenced from another stack. Using PublicSubnetA as an example, and assuming a stack name of "my-network", the exported value would be "my-network-PublicSubnetA". Another stack could use "!ImportValue my-network-PublicSubnetA" to reference this resource as easily as "!Ref" is used within the stack. Often, the initial (base) stack is used as an input parameter, so the stack-name portion can be dynamic, as in:

```
    Fn::ImportValue: !Sub ${BaseStack}-PublicSubnetA
```

...where "BaseStack" is an input parameter of a secondary stack. Irritatingly, !Sub inside an !ImportValue is invalid YAML, so we have to use the "long form" of the function name, Fn::ImportValue.

The Export/Name / !ImportValue techniques shown here are common in multi-team environments. Each team often needs to reference resources from stacks produced by other teams, and in turn produce resources to be referenced by other stacks. The export names become well-known, dependable points of inter-team communication. Further, CloudFormation keeps track of these inter-stack references to prevent a delete or update of one stack from invalidating dependent resources in another.

Summary

Except for the addition of the EC2 NAT instance option, and the bulky Output section, we've only changed a small number of lines from the original template. Parameters and conditions are a compact way to make our template much more capable. We can now create VPCs having from one to six subnets with a variety of permutations possible. The stacks we create can be referenced by other stacks in a highly organized and reliable way. But even more amazing, we can use this template to modify the resulting stack to add or remove subnets, such as when an initial POC grows to become a trial deployment. You can expand upon the techniques here to make this template more sophisticated, such as a VPC with only private subnets.

About the Author

Ken Krueger's professional mission statement is "guiding organizations and individuals toward commercial success through the application of modern technology". He has over has 30 years of experience as a software developer, project leader, project manager, scrum master and instructor spanning the mainframe, client-server, and web eras. He is heavily experienced in Java, Spring, SQL, web development, cloud, and related technologies. Industry experience includes telecommunications, finance, real estate, retail, power generation, shipping, hospitality, and software development. He has an MIS degree from the University of South Florida, an MBA from the Crummer Graduate School of Business at Rollins College, plus scrum master, PMP, AWS, and Java certifications.

Discuss