Building a VPC with CloudFormation - Part 1

Leia em portuguÃªs

Key Takeaways

- AWS CloudFormation allows us to implement "Infrastructure as Code" in an AWS environment.
- A sophisticated Virtual Private Cloud (VPC) is easy to create and update in an automated way with CloudFormation.
- We can re-use CloudFormation templates to build various stacks of resources for various purposes.
- CloudFormation provides pseudo-parameters and intrinsic functions.
- We can delete stacks when we are done with them.

You can find part 2 of this article here.

This article describes how you can use AWS CloudFormation to create and manage a Virtual Private Cloud (VPC), complete with subnets, NATting, route tables, etc. The emphasis is use of CloudFormation and Infrastructure as Code to build and manage resources in AWS, less about the issues of VPC design.

Network Experts: Since the focus will be primarily on CloudFormation, you may have differing opinions from mine regarding CIDR blocks, route tables, etc. That's fine, you can alter this template as needed for your purposes.

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

Why CloudFormation?

You may be wondering why we would use CloudFormation to build our VPC when we can create one via the VPC wizard in the management console. There are several reasons why:

Infrastructure as Code: CloudFormation allows us to create a "stack" of "resources" in one step. Resources are the things we create (EC2 Instances, VPCs, subnets, etc.), a set of these is called a stack. We can write a template that can easily stand up a network stack exactly as we like it in one step. This is faster, repeatable, and more consistent than manually creating our network via the management console or CLI. We can check our template into source control and use it any time we like for any purpose we want.

Updateable: We can modify our network stack by making changes to our CloudFormation template, then modifying the stack on the basis of our revised template. CloudFormation is smart enough to change the stack to match the template.

Reusable: We can reuse this template to create multiple networks in various regions for various purposes and different times.

Drift Detection: A new feature (as of November 2018), CloudFormation can let us know if the resources have “drifted” away from their original configuration. This can happen when administrators manually change resources, which is something mature organizations typically discourage.

Disposable: We can easily delete the stack when we are done with it.

What you need

To follow this tutorial you will need:

An AWS account. You may be able to follow along as an IAM user, however you must have permissions to create VPCs, subnets, route tables, EC2 instances, etc. My general recommendation is to create your own personal AWS account that you completely control. Within this account create an IAM user with full permissions and use it on a day-by-day basis. The costs associated with this tutorial will be extremely light, especially if you delete the stack when finished.

A text editor. Almost anything will do: Sublime, Atom, nano, full-blown IDEs like Eclipse, IntelliJ, Visual Studio, etc. etc. Just make sure you do NOT use a word processor - it will embed special characters which will cause syntax errors. I'll be using Visual Studio Code, my favorite editor (this week).

Beginning

Create an Empty YML file: Start by creating an empty file. Save it under the name "MyNetwork.YML". You could use any name you like, but I'll refer to this file name later. Be sure to give it a YML extension. CloudFormation supports JSON or YAML, and we'll be using the latter. The primary reason is 1) less syntactical ceremony and 2) the ability to PUT COMMENTS in my work, without which I can't remember what I was doing a week ago.

Add Boilerplate: Once you have your empty file, copy and paste this structure within: this is the boilerplate needed for any CloudFormation template:

```
AWSTemplateFormatVersion: 2010-09-09
# This CloudFormation template deploys a basic VPC / Network.
Resources:
```

Comments: Everything after a "#" character is a comment. I encourage you to write your own comments as you make your own discoveries. My suggestion: limit comments to describing anything unusual that you have to do to make something work, especially something that has taken you a while to discover. Imagine a co-worker reading your template and having questions that cannot be answered via published documentation, THAT is what you comment.

Adding a VPC

The first resource to add will be the VPC itself. Copy these lines so your resource section looks like this:

```
Resources:
  # First, a VPC:
  VPC:
    Type: AWS::EC2::VPC
    Properties:
      CidrBlock: 10.1.0.0/16
      EnableDnsSupport: true
      EnableDnsHostnames: true
      Tags:
      - Key: Name
        Value:  !Join ['', [!Ref "AWS::StackName", "-VPC" ]]
```

Indentation: First order of business, get the spacing right. "Resources:" should have no spaces preceding it. "VPC:" should be indented two spaces, "Type" and "Properties" should be indented two more spaces, etc. DO NOT use tabs, unless your editor converts tabs to twin spaces. Most of the editors listed above do, once they understand the kind of file you are creating. Indentation is critical in YAML, it's the price we pay for avoiding JSON's brackets and commas.

VPC: This resource instructs CloudFormation to create a VPC resource, along with some essential properties and a name. The first line is simply "VPC" - this is an arbitrary name we give this resource to identify it within the stack. Confusingly, many resource types have separate "name" properties which is NOT the same thing. The former is only for referencing resources within the stack, the latter is the public name of the resulting resource.

Type: AWS::EC2::VPC: If you Google this you will see a link that takes you directly to the CloudFormation documentation on VPC resources. Here you will find a sample that looks quite similar to the one above. Personally I've found the CloudFormation reference pages indispensable. I freely admit to copying & pasting sample snippets as a starting place for my own resources.

Double-Quotes: One thing you will see in the official documentation is unnecessary use of double and single quotes. The value AWS::EC2::VPC does NOT need to be enclosed in double quotes, neither to the "true" values below. My theory is that AWS Documentation team originally copied the YAML examples from JSON, the latter requiring almost everything in quotes.

Properties: Every resource has properties. The documentation will tell you which are required and which are not. The documentation falls short on telling you default values for optional settings, and explaining the meaning and allowed values for most settings. You are generally expected to have full comprehension of the meaning, syntax, and ramifications of all possible values - which no one does. Probably 90% of my development time has been spent on this ancillary research. Ugh.

CIDR Block: I'm giving my VPC a CIDR of 10.1.0.0/16. If you need a detailed explanation of what this value means, see this description. But briefly, I'm giving the VPC a range of over 65,000 possible private IP addresses to use, all of which will begin with "10.1". That's more than enough addresses for most use cases.

DNS Support / Hostnames: These settings are just allowing DNS hostnames to be automatically assigned to the EC2 instances created within the VPC. Without getting into too much detail about DNS, this will provide a name by which we can reach our EC2 instances, rather than just an IP address. Not strictly necessary, but generally useful.

Tags / Key / Value: Our VPC needs a name. Curiously, there is no "name" property to set. So instead, we will use a tag with the "Key" of "Name" and a "Value" of whatever we want to call our VPC. The value will be used as the display name in many (but not all) situations.

The value is where it gets fun. We could simply hard-code a name, but if we were to create two stacks with this template in the same region we would have two VPCs with the same name (which is actually allowed, try it!). This would be unnecessarily confusing, so instead we are going to take a little extra step to dynamically assign a name.

!Join: The "!Join" is what CloudFormation calls an intrinsic function. CloudFormation has about 15 such functions, and we will see several in this article. Briefly, !Join is used to concatenate or "glue" a text string together. The " within the first bracket identifies the character to place between the concatenated values. We don't want any, so we put an empty string.

!: When reading the documentation, notice that functions have a long form (fn::Join) and an alternative short form that can be used in YAML (!Join). That's generally what you'll see me use. However be aware, there are some situations where the shortcut form is unavailable, usually in cases where you are nesting one function within another.

!Ref: The next function is the Reference function. It references something defined somewhere else, usually within the template. You will see this used A LOT as resources commonly reference each other.

AWS::StackName - This is what CloudFormation calls a pseudo parameter. It will resolve to the name of the stack when we run the template. There are about 7 or so pseudo parameters available for dynamically determining things like the current region, current user, etc. Use of these pseudo parameters greatly increases the flexibility of our template. In this case, the name of our VPC will echo the CloudFormation stack it belongs to. This is very useful when you have lots of resources created by lots of different stacks - trust me on that one.

-VPC: This is simply the suffix of the name of our VPC. So if we run this template through CloudFormation with the stack name "MyNetwork", our VPC will be tagged with the name "MyNetwork-VPC". This name will be used in many, but not all, of the places where VPCs are listed or displayed.

Running the template

Save your work. We are far from complete, but you can check your work by running this template through CloudFormation. There are two options for doing this, the AWS Management Console or the AWS Command Line Interface (CLI).

From the AWS Management Console

From a browser open the AWS Management Console. Sign in, select any region. Find CloudFormation in the menus, use the search feature if needed. Once in, click "Create Stack". Select the option to upload your own template, and hit next.

Do you have any errors? If so CloudFormation is objecting to a syntax issue. Fix these before proceeding. Typically errors result from use of tabs instead of spaces, spaces in unusual places, = instead of :, or special characters embedded by the editor you are using.

Once you are on the next page, give your stack a name. I'll refer to "MyNetwork" for the remainder of this article, but you can use whatever you like. Continue through the wizard without making any other inputs, and create your stack.

While your stack is running, take a look at the "Events" and "Resources" tabs. The events will show you when your VPC is being created, and when it was successfully created. When the last resource in your stack is created, the stack's status will change to created successfully.

Do you have any errors? The stack will show a status of creation failed. Problems encountered at this point are non-syntactical, generally related to the logic of what you are trying to create. Use the Events tab to find the earliest encountered error. I've found most of the time the message is clear enough to guide me toward the problem, and definitely identifies the resource in question.

An important concept to notice is that if there are any errors encountered while creating your stack, the entire stack (all resources) will be rolled back. This behavior can be overridden but is generally good! It is usually a lot easier to start each run from a clean slate.

If your stack fails, it will still display within the stack list, even though there are no resources in the stack. This is done to give you time to investigate the error. Use the 'delete stack' action when you have determined the problem.

From the CLI:

IF you have the AWS CLI installed, and IF you have already configured it with your access key and secret key (any region), and IF your command prompt is located in the same directory as your YML file, you can run this command:

```
aws cloudformation create-stack --stack-name MyNetwork --template-body
file://MyNetwork.yml
```

`--stack-name`: Can be anything you like, but I'll refer to "MyNetwork" throughout the article.

`--template-body`: The file you have been working on.

If you have any syntactical errors, they will be presented back to you immediately. Fix these before proceeding. Typically errors result from use of tabs instead of spaces, spaces in unusual places, = instead of :, or special characters embedded by the editor you are using.

HOWEVER not all can be discovered by CloudFormation right away. You could encounter a non-syntactical error (e.g. permissions) several minutes into a stack creation. When this happens, you'll have to check the status periodically to detect issues. A convenient alternative is to wait for the stack to finish, or error out, with this function:

```
aws cloudformation wait stack-create-complete --stack-name MyNetwork
```

...and when the stack is complete you can use the following to examine it:

```
aws cloudformation describe-stacks
```

Check the VPC: From within the management console, go to the VPC section. You will find your VPC in the list, with the name you provided. Of course, there is nothing in this VPC, but we will get to that next. In case you are curious, there is no cost for this VPC, or the use of CloudFormation, though this will change below (see the NAT section).

Updating a Stack

Before we go further on our template, let's experiment with updating the stack. One of the fantastic features of CloudFormation is the ability to modify a stack based on changes to a template. To demonstrate this, return to your template and make one or more of the following changes:

- Change the VPC's "enableDns*" settings to false.
- Change the VPC's tag/value to
  `!Join ['', [!Ref "AWS::StackName", "-VPC2" ]]`

Save your changes

From the AWS Management Console

Select the existing stack from the list and choose the "Update Stack" action. Select the option to upload your own template, and hit next. Continue hitting next until you reach the screen that asks you to create a change set. The change set is literally the changes that CloudFormation intends to apply to your resources. CloudFormation can make some changes easily, but some require existing resources to be deleted and re-created. Choose the final option to execute the change set and watch the changes being made.

From the CLI

From the CLI update the stack using this command:

```
aws cloudformation update-stack --stack-name MyNetwork --template-body
file://MyNetwork.yml
```

And as before, you can use the wait command to monitor progress:

```
aws cloudformation wait stack-update-complete --stack-name MyNetwork
```

Our stack is small, so the update should only take a few minutes. It is possible you may encounter an error, if this happens the result is even more fascinating as CloudFormation reverts your stack to its previous form. Errors can happen for various reasons, such as when a delete / recreate of a resource would affect another resource outside the stack.

Delete the Stack

One cloud-native concept to get comfortable with is that *Resources are Disposable*. We can create and delete the stack whenever we wish now that we have our template.

From the console, select the stack and take the 'delete stack' action. Or from the CLI run the `aws cloudformation delete-stack --stack-name MyNetwork` command.

Internet Gateway and Attachment

Returning to our template: Most VPCs need to connect to the internet. We'll do this by adding resources for InternetGatway and GatewayAttachment. Copy these lines below your VPC resource:

```
  # Our VPC will need internet access:
  InternetGateway:
    Type: AWS::EC2::InternetGateway
```

```
    DependsOn: VPC
  AttachGateway:
    Type: AWS::EC2::VPCGatewayAttachment
    # Notice how you can't attach an IGW to a VPC unless both are created:
    Properties:
      VpcId: !Ref VPC
      InternetGatewayId: !Ref InternetGateway
```

`AWS::EC2::InternetGateway:` Our VPC won't be able to interact with the public internet without it. It's worth noting that you do not have to connect a VPC to the internet, many organizations build VPCs completely detached from the public world, utilizing VPN connections or AWS Direct Connect to connect exclusively to their existing on-premises networks.

`VPCGatewayAttachment:` More interesting is the AttachGateway resource. This is the actual hook between the VPC and the InternetGateway. Notice the properties; the VpcId references the VPC resource defined above, and the InternetGatewayId references our InternetGateway. When CloudFormation runs a stack, it will try to create all resources simultaneously. However the !Ref function implies an ordering, CloudFormation will create the VPC and InternetGateway simultaneously, but both must be complete before it uses them to create the attachment.

!Ref: As mentioned earlier, this is the built-in CloudFormation "reference" function. This is the key way resources refer to each other.

Later when you run this template, you will find that it takes several seconds to create these first three resources, especially the attachment. This delay will become an issue later, we won't be able to do any interaction with the public internet until the attachment is complete. Watch for use of "DependsOn" later to address this issue.

Some Subnets

Now let's a subnet with this code:

```
 PublicSubnetA:
    Type: AWS::EC2::Subnet
    Properties:
      VpcId: !Ref VPC
      CidrBlock: 10.1.10.0/24
      AvailabilityZone: !Select [ 0, !GetAZs ]    # Get the first AZ in the list
      Tags:
```

```
      - Key: Name
        Value: !Sub ${AWS::StackName}-Public-A
```

PublicSubnetA: Again, and arbitrary name for the resource, applicable only within the Stack. Call it whatever you like. The purpose of "A" in the name is the association of this subnet to availability zone "A", (below)

!Ref VPC: Subnets must exist within a VPC, so this is how we associate the two. The VPC will be created first, then this subnet will attach to it.

CidrBlock: This CIDR is a sub-range of the VPC's 10.1.0.0/16. Essentially it says this subnet will hold all addresses beginning with 10.1.10.*. It also means the subnet will only have 256 total addresses available (actually 251 because AWS reserves 5 for its own purposes, see this explanation.) This is a bit small, and I'm committing a fair amount of waste, so in a real-world scenario you may wish to adjust this value. For deep background on CIDR, see the link above in the section on VPC.

Availability Zone: Each subnet is scoped to a specific availability zone. We could simply hard-code a value like "us-east-1a" but this limits our template to N. Virginia. Best practice is to build templates as region-agnostic as possible, so we are determining AZ dynamically:

!GetAZs: Another built-in CloudFormation function. It returns a list of all the availability zones in the current region (i.e. wherever we are running our stack). For example, if we are running in Oregon, the returned list would be {us-west-2a, us-west-2b, us-west-2c}. Critically, the list will always present these AZs in the same order whenever it is called.

!Select: This built-in function selects the specified index from the specified list. 0 is the first item in the list, 1 is the second, etc. So this expression is saying "give me the first AZ in the list". This will result in "us-west-2a" if running in Oregon, "us-east-1a" if running in N. Virginia, etc. Conclusion: our subnet will automatically associate itself with the 'first' AZ in the region it is running in.

Tags: As before, we want to give our subnet a simple name, but no name property is available. Using a "Name" tag provides a name used in many, but not all, of the places where this subnet will be listed or displayed.

!Sub: This is the substitute function. It takes the string that follows and dynamically substitutes any value found within ${} tags. Since AWS::StackName will render "MyNetwork" in our example, the resulting subnet name will be "MyNetwork-Public-A"

You may have noticed this is achieving the same result as the !Join function shown earlier. I've done this intentionally to show you two alternate techniques. Which is better? That's for you to decide. !Join usually works better if we need to insert a delimiter character between the items being concatenated.

More Subnets

Let's make some more subnets following the pattern above:

```
  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
  PrivateSubnetA:
    Type: AWS::EC2::Subnet
    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
    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
```

The differences between these subnets are 1) the logical names 2) the physical names (via the tag values) 3) the CIDR ranges (nothing overlaps) and 4) the assigned availability zone. The result is four subnets, two are "public", two are "private", two in the "A" availability zone, two in the "B" availability zone. Nice.

You can save and run this template now if you like. I recommend iterative development like this to check for errors.

Adding Routing Tables

Despite their names, subnets are only "public" or "private" based on the definitions of their associated routing tables. Routing tables define where traffic can be routed to in a subnet. The topic is a complex one and I will not cover the full details here, see this information on subnet routing for full information.

```
# Some route tables for our subnets:
PublicRouteTable:
  Type: AWS::EC2::RouteTable
  Properties:
    VpcId: !Ref VPC
    Tags:
    - Key: Name
      Value: Public
PublicRoute1:   # Public route table has direct routing to IGW:
  Type: AWS::EC2::Route
  DependsOn: AttachGateway
  Properties:
    RouteTableId: !Ref PublicRouteTable
    DestinationCidrBlock: 0.0.0.0/0
    GatewayId: !Ref InternetGateway
```

PublicRouteTable: Another arbitrary name used only within the stack, but we're attempting to be descriptive here. Route tables must be associated with a VPC. Tagging with a stack prefixed name to make identity easier later.

PublicRoute1: This describes an entry in the route table. The entry is associated with the public route table (!Ref PublicRouteTable) and routes any internet-bound traffic (DestinationCidrBlock: 0.0.0.0/0) to the internet gateway (!Ref InternetGateway). What is not visible is the implicit first entry included on every route table called the "local" route, all trafffic for 10.1.0.0/16 stays within the VPC. See this detailed information on local routes.

DependsOn: AttachGateway: This is the critical bit. An error will occur if we attempt to build a route table entry to an unattached gateway. To eliminate this error, we let CloudFormation know about this dependency. During stack creation it will not attempt to build this route table entry until after the AttachGateway resource is created. It will also wait for the VPC and internet gateway, but those will finish up early.

You may still be wondering why CloudFormation cannot figure this dependency out for itself - it's a good question. CloudFormation definitely knows about any explicit references you make via !Ref or !GetAtt (covered later), but it really can't piece together whether these resources will actually work when you reference them. CloudFormation is simply making API calls on your behalf. You would encounter the same error if you used the CLI to create a route table entry referencing an unattached gateway. This situation arises in one or two other places too, such as when an EC2 instance needs to connect to an unfinished RDS instance.

You can save your work and run this template, and it should be successful. You will notice most of the creation time is spent by the AttachGateway resource.

At this point, your public subnets are usable. If you launch an EC2 instance with a public IP address in one of these subnets, it will be reachable from the public internet. Now the private subnets...

Private Route Table

The private route table will be similar in most respects, except we won't reference the InternetGateway:

```
  # Here is a private route table:
  PrivateRouteTable:
    Type: AWS::EC2::RouteTable
    Properties:
      VpcId: !Ref VPC
      Tags:
      - Key: Name
        Value: Private
  PrivateRoute1:            # Private route table can access web via NAT (created be
```

```
    Type: AWS::EC2::Route
    Properties:
      RouteTableId: !Ref PrivateRouteTable
      DestinationCidrBlock: 0.0.0.0/0
      # Route traffic through the NAT Gateway:
      NatGatewayId: ???
```

NatGatewayId: If you guessed that "???" is not a valid value, you are correct. To explain this, we will first need to take a quick detour and explain this NAT concept (feel free to skip ahead).

What is this NAT business?

NAT stands for Network Address Translation. For a full explanation see this background information on NAT. But in a nutshell: we do not want instances within our private subnets to be reachable from the public internet. But we do want these instances to be able to initiate outbound connections, such as downloads. Plus we want them to be able to do this without having public IP addresses.

A NAT facilitates this. It will have a public IP address and will be associated with a public subnet. Private instances in private subnets will be able to use this to initiate outbound connections. But the NAT will not allow the reverse, a party on the public internet cannot use the NAT to connect to our private instances.

There are two basic choices for NATting (is that a word?) in AWS, an EC2 instance configured as a NAT, or a relatively new AWS feature called a NAT Gateway. We'll use the latter.

Creating a NAT Gateway

Add these lines to create the NAT Gateway:

```
  # A NAT Gateway:
  NATGateway:
    Type: AWS::EC2::NatGateway
    Properties:
      AllocationId: !GetAtt ElasticIPAddress.AllocationId
      SubnetId: !Ref PublicSubnetA
      Tags:
      - Key: Name
```

```
      Value: !Sub NAT-${AWS::StackName}
  ElasticIPAddress:
    Type: AWS::EC2::EIP
    Properties:
      Domain: VPC
```

The NAT Gateway is associated with one of the public subnets. We could (and probably should) create a NAT Gateway for each of our public subnets, but a single gateway keeps things simple for now. The name is dynamically set as before.

AllocationId: The NAT requires a fixed public IP Address. This is supplied by an Elastic IP address, explained below.

!GetAtt: Another implicit function described here. This references specific property / attribute of another resource. Using !Ref here would not work, the NAT Gateway resource needs the allocationId of the Elastic IP address, not the address itself. It's worth noting that the documentation shows the attributes that you can "get" for each resource, it is generally a subset of a resource's properties..

Elastic IP Address: The EIP is a public IP address whose value remains constant regardless of what it is attached to. The full explanation is here, but suffice it to say it is necessary for our NAT.

Cost: Until now, every resource in our stack is free of charge. NAT Gateways are not. They are charged by the hour and how much traffic moves through them, see VPC Pricing. EIPs are unusual in that there is no charge for them as long as they are being used. AWS charges a nominal amount per hour when they are not attached to running resources, this keeps customers from hoarding them. This makes the cost of this stack is around 1 or 2 US pennies per hour, depending on the region you are running in. Cheap, but be sure to delete the stack when finished to keep these pennies from adding up to dollars.

Now that we have a NAT Gateway, we can reference it from our route table. The revised route should look like this:

```
 PrivateRoute1:            # Private route table can access web via NAT (created belo
     Type: AWS::EC2::Route
     Properties:
       RouteTableId: !Ref PrivateRouteTable
       DestinationCidrBlock: 0.0.0.0/0
       # Route traffic through the NAT Gateway:
       NatGatewayId: !Ref NATGateway
```

This route sends internet-bound traffic (DestinationCidrBlock: 0.0.0.0/0) to the NAT gateway (!Ref NATGateway). In turn, the NAT will send it out the Internet Gateway due to its placement in the public subnet. Response traffic received by the NAT is forwarded to the instance that made the request. Again, not shown is the implicit local route described above.

Attaching Route Tables to Subnets

Last piece, we need to associate the subnets to their relevant route tables. The ones we named "public" need to hook up with the public route table, and the "private" ones to the private route table:

```
 # Attach the public subnets to public route tables,
   # and attach the private subnets to private route tables:
   PublicSubnetARouteTableAssociation:
     Type: AWS::EC2::SubnetRouteTableAssociation
     Properties:
       SubnetId: !Ref PublicSubnetA
       RouteTableId: !Ref PublicRouteTable
   PublicSubnetBRouteTableAssociation:
     Type: AWS::EC2::SubnetRouteTableAssociation
     Properties:
       SubnetId: !Ref PublicSubnetB
       RouteTableId: !Ref PublicRouteTable
   PrivateSubnetARouteTableAssociation:
     Type: AWS::EC2::SubnetRouteTableAssociation
     Properties:
       SubnetId: !Ref PrivateSubnetA
       RouteTableId: !Ref PrivateRouteTable
   PrivateSubnetBRouteTableAssociation:
     Type: AWS::EC2::SubnetRouteTableAssociation
     Properties:
       SubnetId: !Ref PrivateSubnetB
       RouteTableId: !Ref PrivateRouteTable
```

AWS::EC2::SubnetRouteTableAssociation: The association resources simply associate subnets to route tables. Without these associations, your subnets will use a "main" route table associated with the VPC, which is not part of our template. Better to have explicit route tables and associations.

Congratulations! At this point you have a 100% functional CloudFormation template for building a typical VPC. Save it for future use, save / expand it as you need. Use it to create a stack using the instructions covered earlier, updating and deleting it as you see fit.

If you like, you can test your VPC by running EC2 instances within it. Instances with public IP addresses attached to the public subnets will be accessible from the public internet. Instances in the private subnets will be unreachable from the public internet, but will be able to make outbound calls. I'll refer you to other articles for this.

Remember the NAT Gateway will cost a penny or two per hour, so best to delete the stack if you won't be using it..

Conclusion

The ability to create, modify, and delete a stack of resources through CloudFormation is a powerful example of the Infrastructure as Code concept. We no longer need to use screens or commands to manually setup infrastructure now that we have a faster, repeatable, reusable system.

In my next article, I'll show you how to make this template more flexible, using parameters and conditions to create a varying number of subnets, making private subnets optional, and exploring other NAT options. We'll also see how outputs can allow the resources produced by this stack can be easily consumable as inputs to other stacks.

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.

You can find part 2 of this article here.

Discuss