THORN TECHNOLOGIES

THE INFRASTRUCTURE AS CODE HANDBOOK

Best practices, tools, and AWS CloudFormation scripts you can use to implement IaC for your organization

www.thorntech.com

CONTENTS

5 REASONS WHY YOU SHOULD IMPLEMENT INFRASTRUCTURE AS CODE NOW

INFRASTRUCTURE AS CODE CAN HELP AVOID CLOUD DEPLOYMENT INCONSISTENCIES, INCREASE DEVELOPER PRODUCTIVITY, AND LOWER COSTS

There's no doubt that cloud computing has had a major impact on how companies build, scale, and maintain technology products. The ability to click a few buttons to provision servers, databases, and other infrastructure has led to an increase in developer productivity we've never seen before.

While it's easy to spin up simple cloud architectures, mistakes can easily be made provisioning complex ones. Human error will always be present, especially when you can launch cloud infrastructure by clicking buttons on a web app.

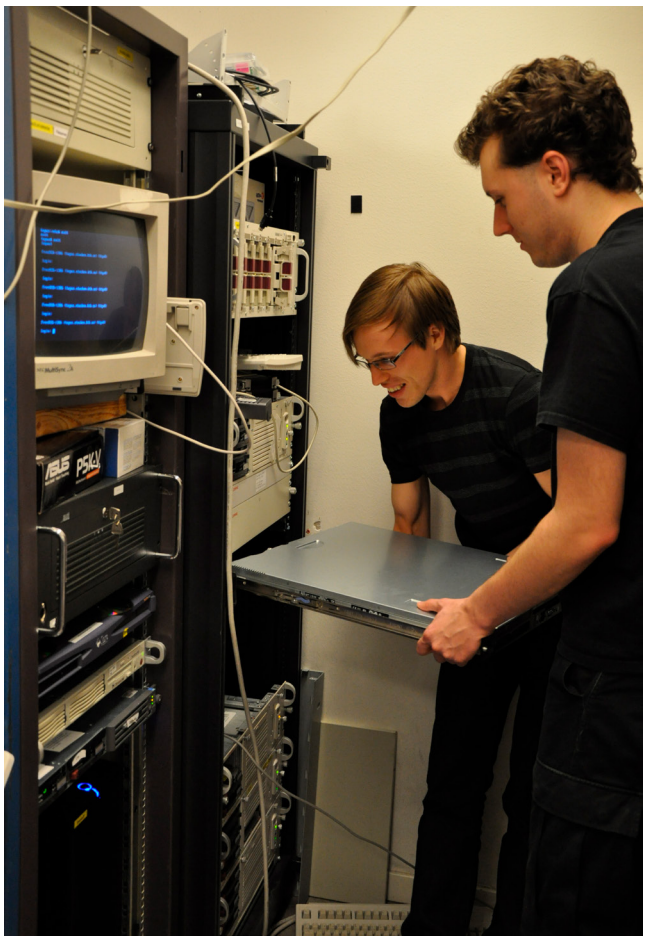
The only way to avoid these kinds of errors is through automation, and Infrastructure as Code is helping engineers automatically launch cloud environments quickly and without mistakes.

HOW IT INFRASTRUCTURE HAS HISTORICALLY BEEN PROVISIONED

In the past, setting up IT infrastructure has been a very manual process.

Humans have to physically rack and stack servers. Then this hardware has to be manually configured to the requirements and settings of the operating system used and application that's being hosted. Finally, the application has to be deployed to the hardware.

Only then can your application be launched. Ugh.



There are many drawbacks to this manual process.

First, it can take a long time to acquire the necessary hardware. You're at the mercy of the hardware manufacturer's production and delivery schedules. And if you need products customized for specific requirements, it could take months to receive your order.

Next, people have to be hired to perform the tedious setup work. You'll need network engineers to set up physical network infrastructure, storage engineers to maintain physical drives, and many others to maintain all of this hardware. That leads to more overhead, management, and costs.

Real estate has to be acquired to build data centers to house all of this hardware. On top of that, you'll have to maintain these data centers, which means paying maintenance and security employees, HVAC and electricity expenses, and many other costs.

Achieving high availability of your applications is big problem as well. You would have to build a backup data center, which could double your real estate and other costs mentioned above.

It would also take a long time to scale an application to accommodate high traffic. Because racking, stacking, and configuring servers is such a slow process, many applications would buckle under spikes in usage while this hardware was being set up. This would have a huge impact in your company's ability to serve your customers and launch new products and services quickly.

To account for these traffic spikes, you may have to provision more servers than you actually need on a daily basis. Thus, you'll have servers that sit idle for large amounts of time, which will increase your costs for this unused capacity.

Finally, because different people are manually deploying these servers, setups are bound to be inconsistent. This can lead to unwanted variance in configurations, which can be detrimental to how your applications run.

The advent of cloud computing addresses some of these problems.

You no longer have to rack and stack servers, which alleviates all of the issues and costs that come with human capital and real estate.

Also, you could spin up servers, databases, and other necessary infrastructure very quickly, which would address the scalability, high availability, and agility problems.

But the configuration consistency issue, where manual setup of cloud infrastructure can lead to discrepancies, still remains. That's where Infrastructure as Code comes into play.

> *"Infrastructure as code is an approach to managing IT infrastructure for the age of cloud, microservices and continuous delivery."*
>
> *– Kief Morris, head of continuous delivery for ThoughtWorks Europe*

DEFINITION OF INFRASTRUCTURE AS CODE

Infrastructure as Code (IaC) is a method to provision and manage IT infrastructure through the use of source code, rather than through standard operating procedures and manual processes.

You're basically treating your servers, databases, networks, and other infrastructure like software. And this code can help you configure and deploy these infrastructure components quickly and consistently.

IaC helps you automate the infrastructure deployment process in a repeatable, consistent manner, which has many benefits.

BENEFITS OF INFRASTRUCTURE AS CODE

Speed and simplicity

IaC allows you to spin up an entire infrastructure architecture by running a script.

Not only can you deploy virtual servers, but you can also launch pre-configured databases, network infrastructure, storage systems, load balancers, and any other cloud service that you may need.

You can do this quickly and easily for development, staging, and production environments, which can make your software development process much more efficient (more about this later).

Also, you can easily deploy standard infrastructure environments in other regions where your cloud provider operates so they can be used for backup and disaster recovery.

You can do all this by writing and running code.

Configuration consistency
Standard operating procedures can help maintain some consistency in the infrastructure deployment process. But human error will always rear its ugly head, which may leave you with subtle differences in configurations that may be difficult to debug.

IaC completely standardizes the setup of infrastructure so there is reduced possibility of any errors or deviations. This will decrease the chances of any incompatibility issues with your infrastructure and help your applications run more smoothly.

Minimization of risk
Imagine having a lead engineer be the only one who knows the ins and outs of your infrastructure setup. Now imagine that engineer leaving your company.

What would you do then? There'd be a bunch of questions, some fear and panic, and many attempts at reverse engineering.

Not only does IaC automate the process, but it also serves as a form of documentation of the proper way to instantiate infrastructure and insurance in the case where employees leave your company with institutional knowledge.

Configurations are bound to change to accommodate new features, additional integrations, and other edits to the application's source code. If an engineer edits the deployment protocol, it can be difficult to pin down exactly what adjustments were made and who was responsible.

Because code can be version-controlled, IaC allows every change to your server configuration to be documented, logged, and tracked. And these configurations can be tested, just like code.

So if there is an issue with the new setup configuration, it can be pinpointed and corrected much more easily, minimizing risk of issues or failure.

Increased efficiency in software development
Developer productivity drastically increases with the use of IaC. Cloud architectures can be easily deployed in multiple stages to make the software development life cycle much more efficient.

Developers can launch their own sandbox environments to develop in. QA can have a copy of production that they can thoroughly test. Security and user acceptance testing can occur in separate staging environments. And then the application code and infrastructure can be deployed to production in one move.

Infrastructure as Code allows your company to use Continuous Integration and Continuous Deployment techniques while minimizing the introduction of human errors after the development stage.

You can also include in your IaC script the spinning down of environments when they're not in use. This will shut down all the resources that your script created, so you won't end up with a bunch of orphan cloud components that everyone is too afraid to delete. This will further increase the productivity of your engineering staff by having a clean and organized cloud account.

Cost savings
Automating the infrastructure deployment process allows engineers to spend less time performing manual work, and more time executing higher-value tasks. Because of this increased productivity, your company can save money on hiring costs and engineers' salaries.

As mentioned earlier, your IaC script can automatically spin down environments when they're not in use, which will further save on cloud computing costs.

CONCLUSION

Infrastructure as Code can simplify and accelerate your infrastructure provisioning process, help you avoid mistakes and comply with policies, keep your environments consistent, and save your company a lot of time and money.

Your engineers can be more productive on focus on higher-value tasks.

And you can better serve your customers.

If IaC isn't something you're doing now, maybe it's time to start!

6 BEST PRACTICES TO GET THE MOST OUT OF IAC

APPLY THESE BEST PRACTICES TO GET THE MOST OUT OF YOUR IAC EFFORTS

In the previous chapter, we highlighted the five primary benefits of automating your infrastructure deployment with IaC. In this chapter, we're going to identify some best practices that can be implemented so you can get the most out of IaC.

These best practices include:

1) Codify everything

What would Infrastructure as Code be without the "code" part?

All infrastructure specifications should be explicitly coded in configuration files, such as AWS CloudFormation templates, Chef recipes, Ansible playbooks, or any other IaC tool you're using (more info about IaC tools can be found in the chapter titled "15 IaC tools you can use to automate your deployments").

These configuration files represent the single source of truth of your infrastructure specifications and describe exactly what cloud components you'll use, how they relate to one another, and how the entire environment is configured.

Infrastructure can then be deployed quickly and seamlessly, and ideally no one should log into a server to manually make adjustments.

Codify all the infrastructure things!

2) Document as little as possible

Your IaC code will essentially be your documentation, so there shouldn't be many additional instructions for your IT employees to execute.

In the past, if any infrastructure component was updated, documentation needed to be updated in lockstep to avoid inconsistencies. We know that this didn't always happen.

With IaC, the code itself represents the documentation of the infrastructure and will always be up to date. Pretty powerful, huh?

Additional documentation, such as diagrams and other setup instructions, may be necessary to educate those employees who are less familiar with the infrastructure deployment process. But most of the deployment steps will be performed by the configuration code, so this documentation should ideally be kept to a minimum.

3) Maintain version control

These configuration files will be version-controlled. Because all configuration details are written in code, any changes to the codebase can be managed, tracked, and reconciled.

Just like with application code, source control tools like Git, Mercurial, Subversion, or others should be used to maintain versions of your IaC codebase. Not only will this provide an audit trail for code changes, it will also provide the ability to collaborate, peer-review, and test IaC code before it goes live.

Code branching and merging best practices should also be used to further increase developer collaboration and ensure that updates to your IaC code are properly managed.

4) Continuously test, integrate, and deploy

Continuous testing, integration, and deployment processes are a great way to manage all the changes that may be made to your infrastructure code.

Testing should be rigorously applied to your infrastructure configurations to ensure that there are no post-deployment issues. Depending on your needs, an array of test types – unit, regression, integration and many more – should be performed. Automated tests can be set up to run each time a change is made to your configuration code.

Security of your infrastructure should also be continuously monitored and tested. DevSecOps is an emerging practice where security professionals work alongside developers to continuously incorporate threat detection and security testing throughout the software development life cycle instead of just throwing it in at the end.

By increasing collaboration between testing, security, and development, bugs and threats can be identified earlier in the development life cycle and thus minimized upon going live.

With a sound continuous integration process, these configuration templates can be provisioned multiple times in different environments such as Dev, Test, and QA. Developers can then work within each of these environments with consistent infrastructure configurations. This continuous integration will mitigate the risk of errors being present that may detrimental to your application when the infrastructure is deployed to production.

5) Make your infrastructure code modular

Microservices architecture, where software is built by developing smaller, modular units of code that can be deployed independently of the rest of a product's components, is a popular trend in the software development world.

The same concept can be applied to IaC. You can break down your infrastructure into separate modules or stacks then combine them in an automated fashion.

There are a few benefits to this approach.

First, you can have greater control over who has access to which parts of your infrastructure code. For example, you may have junior engineers who aren't familiar with or don't have expertise of certain parts of your infrastructure configuration. By modularizing your infrastructure code, you can deny access to these components until the junior engineers get up to speed.

Also, modular infrastructure naturally limits the amount of changes that can be made to the configuration. Smaller changes make bugs easier to detect and allow your team to be more agile.

And if you use the microservices development approach, a configuration template can be created for each microservice to ensure infrastructure consistency. Then all of the microservices can be connected through HTTP or messaging interfaces.

If you'd like to learn more about modularized Infrastructure as Code, check out this Slideshare.

6) Make your infrastructure immutable (when possible)

The idea behind immutable infrastructure is that IT infrastructure components are replaced for each deployment, instead of changed in-place.

You can write code for and deploy an infrastructure stack once, use it multiple times, and never change it. If you need to make changes to your configuration, you would just terminate that stack and build a new one from scratch.

Making your infrastructure immutable provides consistency, avoids configuration drift, and restricts the impact of undocumented changes to your stack. It also improves security and makes troubleshooting easier due to the lack of configuration edits.

While immutable infrastructure is currently a hotly-debated topic, we believe that you should try to make your infrastructure immutable whenever possible to increase the consistency of your configurations.

CONCLUSION

Infrastructure as Code can provide many benefits such as consistency, speed of deployment, simplicity, and more. And if you follow these best practices, you'll get the most out your IaC deployments.

15 INFRASTRUCTURE AS CODE TOOLS YOU CAN USE TO AUTOMATE YOUR DEPLOYMENTS

AUTOMATE YOUR INFRASTRUCTURE DEPLOYMENTS AND CONFIGURATIONS WITH THESE INFRASTRUCTURE AS CODE TOOLS

There are MANY tools that can help you automate your infrastructure. This chapter highlights a few of the more popular tools out there and some of their differentiating features.

CONFIGURATION ORCHESTRATION VS. CONFIGURATION MANAGEMENT

The first thing that should be clarified is the difference between "configuration orchestration" and "configuration management" tools, both of which are considered IaC tools and are included on this list.

Configuration orchestration tools, which include Terraform and AWS CloudFormation, are designed to automate the deployment of servers and other infrastructure. Configuration management tools like Chef, Puppet, and the others on this list help configure the software and systems on this infrastructure that has already been provisioned.

Configuration orchestration tools do some level of configuration management, and configuration management tools do some level of orchestration. Companies can and many times use both types of tools together.

All right, on to the tools!

15 INFRASTRUCTURE AS CODE TOOLS

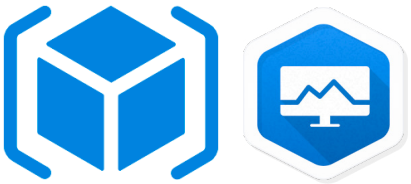
Terraform

Terraform is an infrastructure provisioning tool created by Hashicorp. It allows you to describe your infrastructure as code, creates "execution plans" that outline exactly what will happen when you run your code, builds a graph of your resources, and automates changes with minimal human interaction. Terraform uses its own domain-specific language (DSL) called Hashicorp Configuration Language (HCL). HCL is JSON-compatible and is used to create these configuration files that describe the infrastructure resources to be deployed. Terraform is cloud-agnostic and allows you to automate infrastructure stacks from multiple cloud service providers simultaneously and integrate other third-party services. You even can write Terraform plugins to add new advanced functionality to the platform.

AWS CloudFormation

Similar to Terraform, AWS CloudFormation is a configuration orchestration tool that allows you to code your infrastructure to automate your deployments. Primary differences lie in that CloudFormation is deeply integrated into and can only be used with AWS, and CloudFormation templates can be created with YAML in addition to JSON. CloudFormation allows you to preview proposed changes to your AWS infrastructure stack and see how they might impact your resources, and manages dependencies between these resources. To ensure that deployment and updating of infrastructure is done in a controlled manner, CloudFormation uses Rollback Triggers to revert infrastructure stacks to a previous deployed state if errors are detected. You can even deploy infrastructure stacks across multiple AWS accounts and regions with a single CloudFormation template. And much more.

Azure Resource Manager and Google Cloud Deployment Manager



If you're using Microsoft Azure or Google Cloud Platform, these cloud service providers offer their own IaC tools similar to AWS CloudFormation. Azure Resource Manager allows you to define the infrastructure and dependencies for your app in templates, organize dependent resources into groups that can be deployed or deleted in a single action, control access to resources through user permissions, and more. Google Cloud Deployment Manager offers many similar features to automate your GCP infrastructure stack. You can create templates using YAML or Python, preview what changes will be made before deploying, view your deployments in a console user interface, and much more.

Chef

Chef is one of the most popular configuration management tools that organizations use in their continuous integration and delivery processes. Chef allows you to create "recipes" and "cookbooks" using its Ruby-based DSL. These recipes and cookbooks specify the exact steps needed to achieve the desired configuration of your applications and utilities on existing servers. This is called a "procedural" approach to configuration management, as you describe the procedure necessary to get your desired state. Chef is cloud-agnostic and works with many cloud service providers such as AWS, Microsoft Azure, Google Cloud Platform, OpenStack, and more.

Puppet

Similar to Chef, Puppet is another popular configuration management tool that helps engineers continuously deliver software. Using Puppet's Ruby-based DSL, you can define the desired end state of your infrastructure and exactly what you want it to do. Then Puppet automatically enforces the desired state and fixes any incorrect changes.

This "declarative" approach – where you declare what you want your configuration to look like, and then Puppet figures out how to get there – is the primary difference between Puppet and Chef. Also, Puppet is mainly directed toward system administrators, while Chef primarily targets developers. Puppet integrates with the leading cloud providers like AWS, Azure, Google Cloud, and VMware, allowing you to automate across multiple clouds.

Saltstack

Saltstack differentiates itself from tools like Chef and Puppet by taking an "infrastructure as data" approach, instead of "infrastructure as code." What this means is that Saltstack's declarative configuration patterns, while written in Python, are language-agnostic (i.e. you don't need to learn a specific DSL to create them) and thus are more easily read and understood. Another differentiator is that Saltstack supports remote execution of commands, whereas Chef and Puppet's configuration code needs to be pulled from their servers.

Ansible

Ansible is an infrastructure automation tool created by Red Hat, the huge enterprise open source technology provider. Ansible models your infrastructure by describing how your components and system relate to one another, as opposed to managing systems independently. Ansible doesn't use agents, and its code is written in YAML in the form of Ansible Playbooks, so configurations are very easy to understand and deploy. You can also extend Ansible's functionality by writing your own Ansible modules and plugins.

Juju

Juju is an IaC tool brought to you by Canonical, the company behind Ubuntu. You can create Juju charms, which are sets of scripts that deploy and operate software, and bundles, which are collections of charms linked together to deploy entire app infrastructures all at once. You can then use Juju to manage and apply changes to your infrastructure with simple commands. Juju works with bare metal, private clouds, multiple public cloud providers, as well as other orchestration tools like Puppet and Chef.

Docker

Docker helps you easily create containers that package your code and dependencies together so your applications can run in any environment, from your local workstation to any cloud service provider's servers. YAML is used to create configuration files called Dockerfiles. These Dockerfiles are the blueprints to build the container images that include everything – code, runtime, system tools and libraries, and settings – needed to run a piece of software.

Because it increases the portability of applications, Docker has been especially valuable in organizations who use hybrid or multi-cloud environments. The use of Docker containers has grown exponentially over the past few years and many consider it to be the future of virtualization.

Vagrant

Vagrant is another IaC tool built by HashiCorp, the makers of Terraform. The difference is that Vagrant focuses on quickly and easily creating development environments that use a small amount of virtual machines, instead of large cloud infrastructure environments that can span hundreds or thousands of servers across multiple cloud providers. Vagrant runs on top of virtual machine solutions from VirtualBox, VMware, AWS, and any other cloud provider, and also works well with tools like Chef and Puppet.

Pallet

Pallet is an IaC tool used to automate infrastructure in the cloud, on server racks, or virtual machines, and provides a high level of environment customization. You can run Pallet from anywhere, and you don't have to set up and maintain a central server. Pallet is written in Clojure, runs in a Java Virtual Machine, and works with AWS, OpenStack, VirtualBox, and others, but not Azure nor GCP. You can use Pallet to start, stop, and configure nodes, deploy projects, and even run administrative tasks.

(R)?ex

(R)?ex is an open-source, weirdly-spelled infrastructure automation tool. "(R)?ex" is too hard to type over and over again, so I'm going to spell it "Rex" from now on. Rex has its own DSL for you to describe your infrastructure configuration in what are called Rexfiles, but you can use Perl to harness Rex's full power. Like Ansible, Rex is agent-less and uses SSH to execute commands and manage remote hosts. This makes Rex easy to use right away.

CFEngine

CFEngine is one of the oldest IaC tools out there, with its initial release in 1993. CFEngine allows you to define the desired states of your infrastructure using its DSL. Then its agents monitor your environments to ensure that their states are converging toward the desired states, and reports the outcomes. It's written in C and claims to be the fastest infrastructure automation tool, with execution times under 1 second.

NixOS

NixOS is a configuration management tool that aims to make upgrading infrastructure systems as easy, reliable, and safe as possible. The platform does this by making configuration management "transactional" and "atomic." What this means is that if an upgrade to a new configuration is interrupted for some reason, the system will either boot up in the new or old configuration, thus staying stable and consistent. NixOS also makes it very easy to rollback to a prior configuration, since new configuration files don't overwrite old ones. These configuration files are written in Nix expression language, its own unique functional language.

CONCLUSION

So there you have it. Check out these configuration orchestration and management tools that you can use to implement Infrastructure as Code and help you automate your infrastructure.

This list is by no means exhaustive but it should give you a starting point for tools that you can use during your IaC journey.

WHAT IS AWS CLOUDFORMATION AND HOW CAN IT HELP YOUR IAC EFFORTS?

CLOUDFORMATION IS A POWERFUL INFRASTRUCTURE AS CODE TOOL THAT CAN HELP AUTOMATE AND MANAGE YOUR AWS DEPLOYMENTS

The last chapter highlighted 15 popular IaC tools you can use to automate your deployments. One of the tools we wrote about is CloudFormation, which is the IaC tool offered by the largest cloud service provider in the world, Amazon Web Services.

In 2017, over 350,000 AWS customers used AWS CloudFormation to deploy and manage over 2.4 million infrastructure stacks. Impressive.

In this chapter, we'll take a look at what CloudFormation's primary features are, how it can be used with other IaC tools, and what companies use it.

WHAT IS CLOUDFORMATION?

AWS CloudFormation is a configuration orchestration tool that allows you to codify your infrastructure to automate your deployments.

CloudFormation templates can be created with YAML in addition to JSON. Or you can use AWS CloudFormation Designer to visually create your templates and see the interdependencies of your resources.

CloudFormation takes a declarative approach to configuration, meaning that you tell it what you want your environment to look like, and it finds its way there.

During this configuration process, CloudFormation automatically manages dependencies between your resources. Thus, you don't have to specify the order in which resources are created, updated, or deleted. CloudFormation automatically determines the correct sequence of actions to create your environment, though you can use the DependsOn attribute, wait condition handlers, and nested stacks to specify the order of operations, if necessary.

Sometimes updating an infrastructure stack can cause anxiety because you're not sure what changes might break the environment. Not to fear! CloudFormation Change Sets allow you to preview how your resources will be impacted before any changes are executed. Only after you execute your change set will your stack be edited.

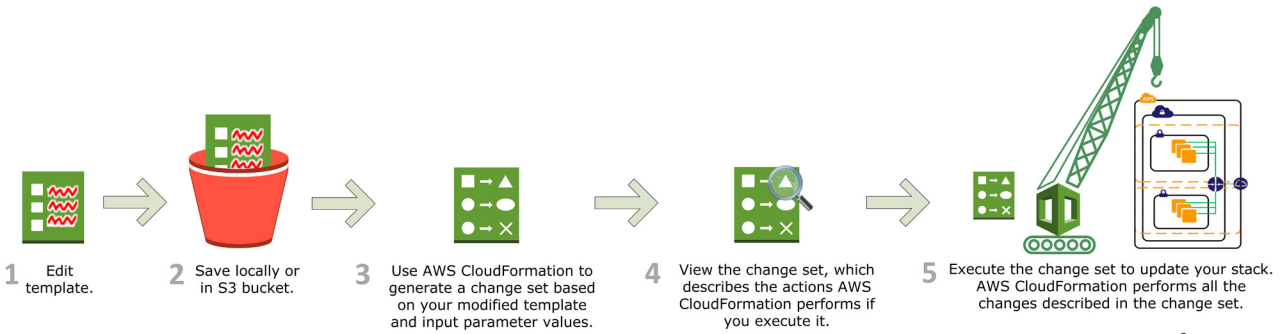


Image courtesy of AWS

Even if you execute a change set that has errors in it, CloudFormation has Rollback Triggers that allow you to monitor your stack creation or update process and roll back your environment to a previous state. You can specify thresholds to monitor in CloudWatch and integrate them into your CloudFormation templates. When these thresholds are exceeded, the Rollback Triggers revert your environment back to the previously deployed state.

CloudFormation StackSets allow you to deploy, update, or delete infrastructure stacks across multiple AWS regions and accounts with a single CloudFormation template. Before StackSets existed, every infrastructure environment had to be deployed independently, and custom scripts had to be written to deploy these stacks to multiple accounts and regions. StackSets now make it much easier to maintain consistency when you add new regions and accounts.

And Custom Resources let you write custom provisioning logic in your CloudFormation scripts. For instance, if you'd like to define resources that CloudFormation doesn't support yet, or if you need to create a resource that's specific to your use case, Custom Resources allows you to manage all of this in a single stack.

CloudFormation supports many AWS services. AWS frequently releases additional useful functionality, so the sky's the limit on the things you can do with CloudFormation.

COMBINING CLOUDFORMATION WITH OTHER IAC TOOLS

As mentioned in our prior chapter about IaC tools, CloudFormation is a configuration orchestration tool, which is used to automate the deployment of servers and other infrastructure like databases and load balancers.

CloudFormation is often used in conjunction with configuration management tools, which are designed to configure the software and systems that run on this infrastructure.

This combination provides seamless deployment and configuration of AWS infrastructure and the applications that run on top of it.

Let's say that you want to build a SaaS data analytics application on AWS. You can use a CloudFormation template to provision a web server on EC2, a DynamoDB database, and any other AWS resources you need. Then you can install a configuration management tool to set up the operating systems and software on these EC2 instances.

Chef and Puppet are the most popular configuration management products used with CloudFormation, and AWS has a product called OpsWorks that provides managed instances of these tools.

The combination of CloudFormation with configuration management tools allows you to automate the configuration, deployment, and management of cloud resources and the software that run on them from a single template. This greatly improves the efficiency and consistency of your infrastructure deployments.

COMPANIES WHO USE CLOUDFORMATION

If you use CloudFormation to automate your deployments, you're in good company.

Nextdoor

Nextdoor is a private social network for local neighbors to connect with each other and share community news. The company has over 160,000 neighborhoods across the globe and more than 10 million users on its platform.

Nextdoor uses CloudFormation templates for flexible, one-click server deployment and network creation. They also use Puppet to define and configure the software and operating systems that run on these AWS servers.

Coinbase

Coinbase is the largest consumer Bitcoin wallet in the world. Over 10 million users have purchased or traded over $50 billion worth of cryptocurrencies through their platform.

Most if not all of Coinbase's infrastructure is designed and managed with CloudFormation templates. This allows the company to easily replicate infrastructure stacks for all phases of their development process and provides version control to ensure their environments are configured correctly over time.

Expedia

Expedia Group is one of the largest travel booking companies in the world, with well-known brands such as Expedia.com, Hotels.com, Travelocity, and many more. In 2017, the company had $88.4 billion of gross bookings through its various platforms.

The global company has a multi-region, multi-availability zone architecture, and uses CloudFormation in combination with Chef to deploy its entire front and backend stack into its Amazon Virtual Private Cloud environment.

CONCLUSION

As you can see, AWS is a really powerful tool that many large and influential companies use to deploy and manage their infrastructure stacks.

We primarily work with AWS, so we're all over CloudFormation. And we've built a couple of products on the AWS Marketplace (SFTP Gateway and WP SureStack) using CloudFormation, so we understand the power that it can have in automating AWS infrastructure deployments.

HOW AWS CLOUDFORMATION WORKS

(AND HOW TO CREATE A VIRTUAL PRIVATE CLOUD WITH IT)

HERE'S AN IN-DEPTH WALKTHROUGH OF HOW CLOUDFORMATION WORKS AND AN ANALYSIS OF A TEMPLATE THAT CREATES A VPC

In our last chapter, we provided an overview of AWS CloudFormation and how it can help you manage your deployments. In this chapter, we're going to dig deeper into CloudFormation, provide a template that we wrote that creates a virtual private cloud (VPC), and dissect how it works.

This is the first of three templates that we'll break down, with each getting progressively more complex. In the next two chapters, we'll highlight how we built the CloudFormation templates for our product SFTP Gateway and a Redshift stack. Let's go!

THE ANATOMY OF A CLOUDFORMATION TEMPLATE

Here's a super simple CloudFormation template that creates an S3 bucket:

```
AWSTemplateFormatVersion: 2010-09-09
Description: Bucket Stack
Resources:
  S3Bucket:
    Type: AWS::S3::Bucket
      Properties:
        BucketName: my-bucket
```

You can create CloudFormation templates using JSON or YAML. We prefer the latter, so all the templates included in this article are in YAML. Regardless of which you choose, templates can consist of the primary sections of information highlighted below, with the Resources section being the only one that is mandatory.

FORMAT VERSION (OPTIONAL)

This is the AWS CloudFormation template version that your template conforms to and identifies the capabilities of the template. This is not the same as the API or WSDL version.

At the moment, the latest and only valid template version is 2010-09-09.

While there is only one valid version at the moment and this field is optional, it's a good idea to include it in your templates for future reference.

DESCRIPTION (OPTIONAL)

This section is a text string that provides the reader with a short description (0 to 1024 bytes in length) of the template. This section must come right after the Format Version section.

Be descriptive but concise!

METADATA (OPTIONAL)

This section isn't used all that often, but here you can include additional information about your template. This can include information for third-party tools that you may use to generate and modify these templates and other general data.

Please note that the Metadata section is not to be confused with the Metadata attribute that falls under the Resources section. This Metadata section should include information about the template overall, while the Metadata attribute should include data about a specific resource.

PARAMETERS (OPTIONAL)

You can use the Parameters section to input custom values to your template when you create or update a stack so you can make these templates portable for use in other projects.

For instance, you can create parameters that specify the EC2 instance type to use, an S3 bucket name, an IP address range, and other properties that may be important to your stack.

The Resources and Output sections often refer to Parameters, and these references must be included within the same template.

MAPPINGS (OPTIONAL)

The Mappings section allows you to create key-value dictionaries to specify conditional parameter values.

Examples of this include deploying different AMIs for each AWS region, or mapping different security groups to Dev, Test, QA, and Prod environments that otherwise share the same infrastructure stack.

To retrieve values in a map, you use the "FN::FindInMap" intrinsic function in the Resources and Outputs sections.

CONDITIONS (OPTIONAL)

Conditions allow you to use logic statements (just like an "if then" statement) to declare what should happen under certain situations.

We mentioned an example above about Dev, Test, QA, and Prod environments. In this case, you can use conditions to specify the type of EC2 instance to deploy in each of these environments. If the environment is Prod, you can set the EC2 instance to be m4.large. If the environment is Test, you can set it to be t2.micro to save money.

TRANSFORM (OPTIONAL)

The Transform section allows you to simplify your CloudFormation template by condensing multiple lines of resource declaration code and reusing template components.

There are two types of transforms that CloudFormation supports:

• "AWS::Include" refers to template snippets that reside outside of the main CloudFormation template you're working with. Thus, you can make multi-line resource declarations in YAML or JSON files stored elsewhere and refer to them with a single line of code in your primary CloudFormation template.

• "AWS::Serverless" specifies the version of the AWS Serverless Application Model (SAM) to use and how to process it.

You can declare multiple transforms in a template and CloudFormation executes them in the order specified. You can also use template snippets across multiple CloudFormation templates.

RESOURCES (REQUIRED)

The Resources section is the only section that is required in a CloudFormation template.

In this section, you declare the AWS resources, such as EC2 instances, S3 buckets, Redshift clusters, and others, that you want deployed in your stack. You also specify the properties, such as instance size, IAM roles, and number of nodes, for each of these components.

This is the section that will take up the bulk of your templates.

OUTPUTS (OPTIONAL)

In the Outputs section, you'll describe the values that are returned when you want to view the properties of your stack.

You can export these outputs for use in other stacks, or simply view them on the CloudFormation console or CLI as a convenient way to get important information about your stack's components.

That's a lot of information about CloudFormation sections to digest. All of this will become clearer as we walk through some real-world templates, which we'll do now.

CREATING A VPC WITH AWS CLOUDFORMATION

The template we've provided below creates a virtual private cloud (VPC) with public and private subnets. This will allow you to launch AWS resources in a virtual network that you define and have complete control over.

In this template, we create a VPC with a public subnet for your web servers that are publicly addressable, and a private subnet where your backend components like databases or application servers will reside and be safe from the prying eyes of the internet.

The public subnet will connect to the internet via an Internet Gateway, and a Route Table tells the public subnet how to find the Internet Gateway.

We'll then replicate the public and private subnets in another availability zone for high availability.

This template is pretty straightforward and only contains the Format Version, Description, and Resources sections.

Here's the CloudFormation template in its entirety:

```
AWSTemplateFormatVersion: 2010-09-09
Description: Creates a VPC with public and private subnets
Resources:
  VPC:
    Type: AWS::EC2::VPC
    Properties:
      CidrBlock: 192.168.101.0/24
  PublicSubnetA:
    Type: AWS::EC2::Subnet
    Properties:
      AvailabilityZone: !Select
        - 0
        - !GetAZs
          Ref: AWS::Region
      CidrBlock: 192.168.101.0/28
      MapPublicIpOnLaunch: true
      VpcId: !Ref VPC
  PrivateSubnetA:
    Type: AWS::EC2::Subnet
    Properties:
      AvailabilityZone: !Select
        - 0
        - !GetAZs
          Ref: AWS::Region
      CidrBlock: 192.168.101.32/28
      MapPublicIpOnLaunch: false
      VpcId: !Ref VPC
  InternetGateway:
    Type: AWS::EC2::InternetGateway
  AttachGateway:
    Type: AWS::EC2::VPCGatewayAttachment
    Properties:
      InternetGatewayId: !Ref InternetGateway
      VpcId: !Ref VPC
  PublicRouteTable:
    DependsOn: AttachGateway
    Type: AWS::EC2::RouteTable
```

```
    Properties:
      VpcId: !Ref VPC
  PublicDefaultRoute:
    Type: AWS::EC2::Route
    Properties:
      DestinationCidrBlock: 0.0.0.0/0
      GatewayId: !Ref InternetGateway
      RouteTableId: !Ref PublicRouteTable
  PublicRouteAssociationA:
    Type: AWS::EC2::SubnetRouteTableAssociation
    Properties:
      RouteTableId: !Ref PublicRouteTable
      SubnetId: !Ref PublicSubnetA
  PublicSubnetB:
    Type: AWS::EC2::Subnet
    Properties:
      AvailabilityZone: !Select
        - 1
        - !GetAZs
          Ref: AWS::Region
      CidrBlock: 192.168.101.16/28
      MapPublicIpOnLaunch: true
      VpcId: !Ref VPC
  PrivateSubnetB:
    Type: AWS::EC2::Subnet
    Properties:
      AvailabilityZone: !Select
        - 1
        - !GetAZs
          Ref: AWS::Region
      CidrBlock: 192.168.101.48/28
      MapPublicIpOnLaunch: false
      VpcId: !Ref VPC
  PublicRouteAssociationB:
    Type: AWS::EC2::SubnetRouteTableAssociation
    Properties:
      RouteTableId: !Ref PublicRouteTable
      SubnetId: !Ref PublicSubnetB
```

[You can download this template here.]

CREATION OF THE VPC AND PUBLIC AND PRIVATE SUBNETS

First, we add a VPC that encompasses the entire network we're about to create:

```
Resources:
  VPC:
    Type: AWS::EC2::VPC
    Properties:
      CidrBlock: 192.168.101.0/24
```

Here's what's going on:

- VPC: This is the name you give to the resource.
- Type: This defines the CloudFormation resource type. Check out the resource type documentation to find supported properties.
- Properties: Each resource has required and optional properties. In this case, you can define the IP address range in CIDR notation.

Next we add a public and private subnet:

```
  PublicSubnetA:
    Type: AWS::EC2::Subnet
    Properties:
      AvailabilityZone: !Select
        - 0
        - !GetAZs
          Ref: AWS::Region
      CidrBlock: 192.168.101.0/28
      MapPublicIpOnLaunch: true
      VpcId: !Ref VPC
  PrivateSubnetA:
    Type: AWS::EC2::Subnet
    Properties:
      AvailabilityZone: !Select
        - 0
        - !GetAZs
          Ref: AWS::Region
      CidrBlock: 192.168.101.32/28
      MapPublicIpOnLaunch: false
      VpcId: !Ref VPC
```

As before, you'll notice the same resource structure of name, type, and supported properties. The "AWS::EC2::Subnet" type happens to support a few more properties.

The example also uses some special syntax to make the template a little more dynamic, such as:

Pseudo parameters
A pseudo parameter dynamically resolves to a value, given the context of the CloudFormation stack. For example, "Ref: AWS::Region" gets converted to the region in which you deploy the template (i.e. us-east-1).

Intrinsic functions
A CloudFormation template is configuration, not code. However, you have access to some basic runtime logic.

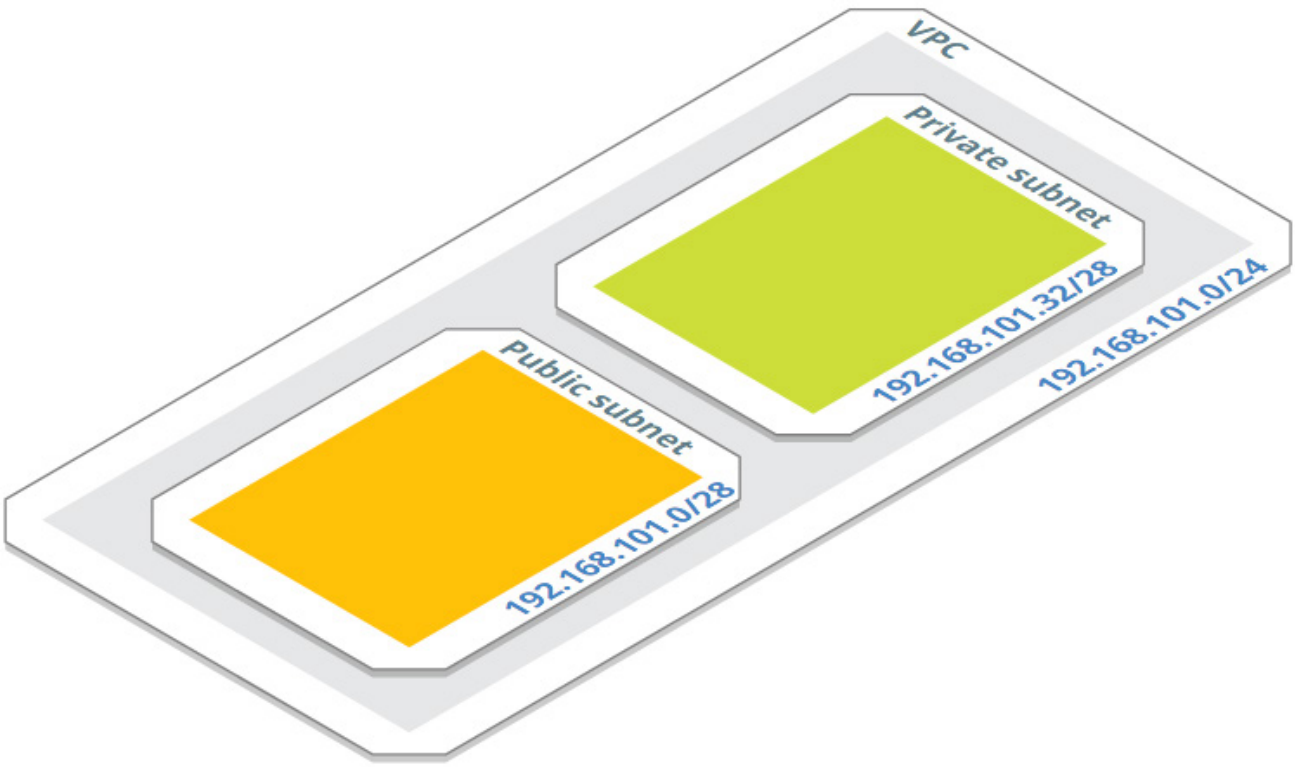
In the above example, "!GetAZs Ref: AWS::Region" gets all the availability zones in the current region. And the "!Select" function gets the first AZ in the list.

Another use of an intrinsic function is "!Ref VPC". This gets the VPC ID of the VPC resource defined earlier.

Here's a breakdown of what's happening in the code above.

- PublicSubnetA is our public subnet, and any EC2 instances provisioned will be given a public IP address.
- PrivateSubnetA is our private subnet.
- Each subnet is provisioned in the first availability zone in the current region. For example, this could be "us-east-1a".
- Each subnet is placed inside the VPC defined earlier.
- Each subnet's IP address range contains 16 addresses. These ranges are non-overlapping, and fall within the overall VPC's IP range.

We wind up with this architecture:



ADDING AN INTERNET GATEWAY AND ROUTE TABLE

Next, we create an Internet Gateway. This allows instances within the VPC to access the internet. And it also allows the internet to access servers within the public subnet:

```
 InternetGateway:
   Type: AWS::EC2::InternetGateway
 AttachGateway:
   Type: AWS::EC2::VPCGatewayAttachment
   Properties:
     InternetGatewayId: !Ref InternetGateway
     VpcId: !Ref VPC
```

Behind the scenes, the Internet Gateway has no way to associate itself with our VPC. So you have to create an "AWS::EC2::VPCGatewayAttachment" resource to perform this task.

Next we create a Route Table, plus a few related resources:

```
PublicRouteTable:
  DependsOn: AttachGateway
  Type: AWS::EC2::RouteTable
  Properties:
    VpcId: !Ref VPC
PublicDefaultRoute:
  Type: AWS::EC2::Route
  Properties:
    DestinationCidrBlock: 0.0.0.0/0
    GatewayId: !Ref InternetGateway
    RouteTableId: !Ref PublicRouteTable
PublicRouteAssociationA:
  Type: AWS::EC2::SubnetRouteTableAssociation
  Properties:
    RouteTableId: !Ref PublicRouteTable
    SubnetId: !Ref PublicSubnetA
```

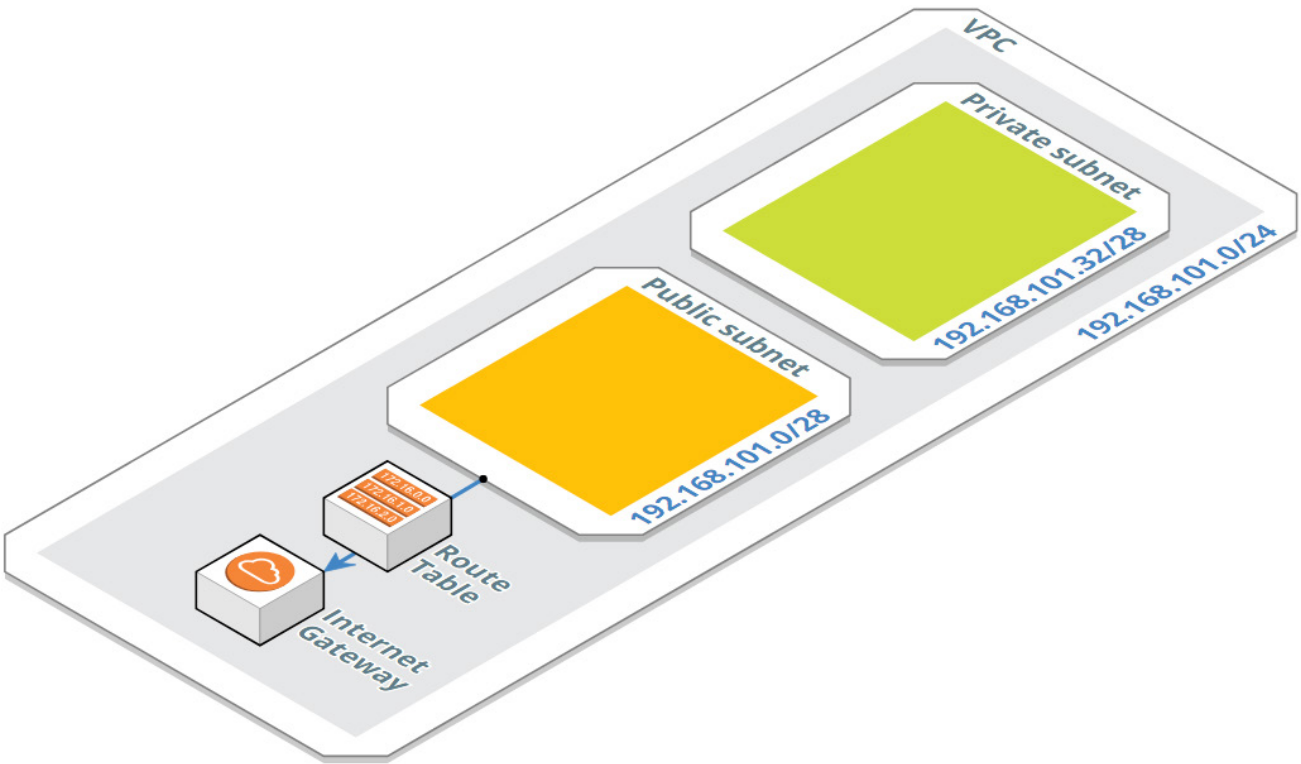
The Route Table only has one property, "VpcId: !Ref VPC", which associates itself with a VPC.

Route ("AWS::EC2::Route") resources contain a single route, which points an IP address range to a gateway. The Route associates itself with the Route Table.

A Subnet Route Table Association wires up the route table to a subnet. In this case, the Route Table directs all traffic from PublicSubnetA to the internet.

One thing to note is the "DependsOn" attribute. CloudFormation is usually pretty good about spinning up dependencies in the correct order. However, you can use the "DependsOn" attribute to explicitly define a dependency.

This architecture shows the addition of the Internet Gateway and Route Table:



REPLICATE SUBNETS FOR HIGH AVAILABILITY

Finally, we replicate the subnets into a new availability zone to facilitate high availability.

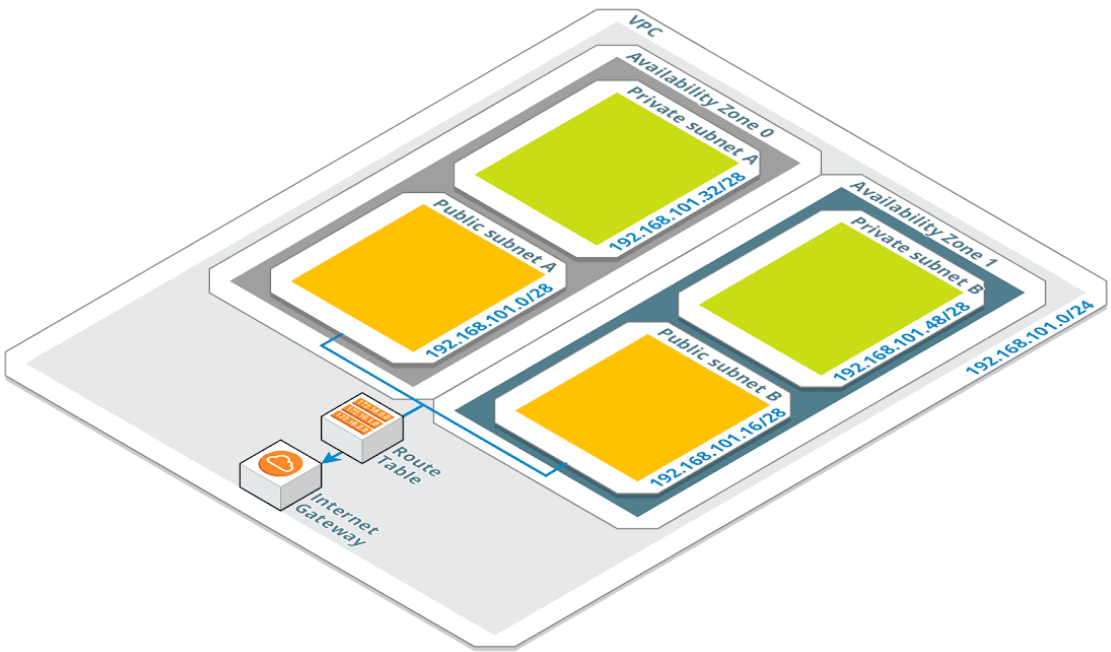
```
  PublicSubnetB:
    Type: AWS::EC2::Subnet
    Properties:
      AvailabilityZone: !Select
        - 1
        - !GetAZs
          Ref: AWS::Region
      CidrBlock: 192.168.101.16/28
      MapPublicIpOnLaunch: true
      VpcId: !Ref VPC
  PrivateSubnetB:
    Type: AWS::EC2::Subnet
```

```
    Properties:
      AvailabilityZone: !Select
        - 1
        - !GetAZs
          Ref: AWS::Region
      CidrBlock: 192.168.101.48/28
      MapPublicIpOnLaunch: false
      VpcId: !Ref VPC
  PublicRouteAssociationB:
    Type: AWS::EC2::SubnetRouteTableAssociation
    Properties:
      RouteTableId: !Ref PublicRouteTable
      SubnetId: !Ref PublicSubnetB
```

This is all quite similar to the what we already created for public and private subnet A. Like before, the IP address ranges are unique and non-overlapping. The main difference with set B is that they get provisioned in the second availability zone rather than the first.

One thing to note is that PublicSubnetB needs its own route association, since it's one association per subnet.

Finally, you wind up with this beautiful VPC architecture:



You now have the network infrastructure to launch EC2 instances, databases, or other AWS resources with some baseline security! You can use this template as a starting point for future projects.

Read on to learn how to use CloudFormation to have EC2, IAM, and S3 work together.

HOW TO INCORPORATE S3, EC2, AND IAM IN A CLOUDFORMATION TEMPLATE

HERE'S HOW YOU CAN USE CLOUDFORMATION TO HAVE EC2, IAM, AND S3 WORK TOGETHER

In our last chapter, we dug deep into how AWS CloudFormation works and provided an analysis of a VPC template we created.

Our next template example is that of SFTP Gateway, a product that we sell on the AWS Marketplace that makes it easy to transfer files via SFTP to Amazon S3. We have over 1000 customers using the product, so it's a useful tool!

We'll incorporate S3, EC2, IAM, Security Groups, and more to facilitate this file transfer.

Here's the template in its entirety:

```
AWSTemplateFormatVersion: 2010-09-09
Mappings:
  RegionMap:
    ap-northeast-1:
      AMI: ami-0d1a9e6b
    ap-northeast-2:
      AMI: ami-0ca50362
    ap-south-1:
      AMI: ami-989fd6f7
    ap-southeast-1:
```

```
      AMI: ami-db7a25b8
    ap-southeast-2:
      AMI: ami-8c14e0ee
    ca-central-1:
      AMI: ami-3d13a859
    eu-central-1:
      AMI: ami-fd6fe192
    eu-west-1:
      AMI: ami-e8922c91
    eu-west-2:
      AMI: ami-09223c6d
    eu-west-3:
      AMI: ami-9563d4e8
    sa-east-1:
      AMI: ami-eb4f0887
    us-east-1:
      AMI: ami-c599febf
    us-east-2:
      AMI: ami-6a1c350f
    us-west-1:
      AMI: ami-99b983f9
    us-west-2:
      AMI: ami-a7ca10df
Resources:
  SFTPGatewayInstance:
    Type: AWS::EC2::Instance
    Metadata:
      AWS::CloudFormation::Init:
        config:
          commands:
            setup:
              command: /usr/local/bin/sftpgatewaysetup
    Properties:
      IamInstanceProfile: !Ref RootInstanceProfile
      ImageId: !FindInMap
        - RegionMap
        - !Ref AWS::Region
        - AMI
      InstanceType: !Ref EC2Type
      BlockDeviceMappings:
```

```
        - DeviceName: /dev/xvda
          Ebs:
            VolumeSize: !Ref DiskVolumeSize
            VolumeType: gp2
      KeyName: !Ref KeyPair
      SecurityGroupIds:
        - !Ref SFTPGatewaySG
      SubnetId: !Ref SubnetID
      Tags:
        - Key: Name
        Value: SFTPGateway Instance
      UserData:
        Fn::Base64: !Sub |
          #!/bin/bash
          yum update -y aws-cfn-bootstrap
          /opt/aws/bin/cfn-init --stack ${AWS::StackName} --resource
SFTPGatewayInstance  --region ${AWS::Region}

 SFTPGatewayBucket:
   DeletionPolicy: Retain
   Type: AWS::S3::Bucket
   Properties:
     BucketName: !Sub sftpgateway-${SFTPGatewayInstance}
   DependsOn:
     - SFTPGatewayInstance
 RootInstanceProfile:
   Type: AWS::IAM::InstanceProfile
   Properties:
     Path: /
     Roles:
       - !Ref S3WritableRole
 S3WritableRole:
   Type: AWS::IAM::Role
   Properties:
     AssumeRolePolicyDocument:
       Version: 2012-10-17
       Statement:
         - Effect: Allow
           Principal:
             Service: ec2.amazonaws.com
           Action: sts:AssumeRole
```

```
      Path: /
  RolePolicies:
    Type: AWS::IAM::Policy
    DependsOn:
      - SFTPGatewayInstance
    Properties:
      PolicyName: SFTPGatewayInstancePolicy
      PolicyDocument:
        Version: 2012-10-17
        Statement:
          - Effect: Allow
            Action: 's3:*'
            Resource: '*'
      Roles:
        - !Ref S3WritableRole
  SFTPGatewaySG:
    Type: AWS::EC2::SecurityGroup
    Properties:
      GroupDescription: SFTPGateway Security Group
      VpcId: !Ref VPCIdName
      SecurityGroupIngress:
        - IpProtocol: tcp
          FromPort: 22
          ToPort: 22
          CidrIp: 0.0.0.0/0
  IPAddress:
    Properties:
      Domain: vpc
      InstanceId: !Ref SFTPGatewayInstance
    Type: AWS::EC2::EIP
Parameters:
  EC2Type:
    Description: SFTPGateway Instance Type
    Type: String
    Default: t2.micro
    AllowedValues:
      - t2.micro
      - t2.small
      - t2.medium
      - t2.large
```

```
        - m3.medium
        - m3.large
        - m3.xlarge
        - m4.large
        - m4.xlarge
        - c3.large
        - c3.xlarge
        - c4.large
        - c4.xlarge
        - r3.large
        - r3.xlarge
    KeyPair:
      Description: EC2 KeyPair
      Type: AWS::EC2::KeyPair::KeyName
      ConstraintDescription: Existing EC2 KeyPair.
    DiskVolumeSize:
      Default: 32
      Description: Disk volume size in GB. Must be at least 32.
      ConstraintDescription: Must be a number greater or equal to 32
      MinValue: 32
      Type: Number
    VPCIdName:
      Description: Select the VPC to launch the SFTPGateway into
      Type: AWS::EC2::VPC::Id
    SubnetID:
      Description: Subnet ID
      Type: AWS::EC2::Subnet::Id
  Outputs:
    ElasticIP:
      Value: !Ref IPAddress
      Description: Elastic IP address
```

You can [download the SFTP Gateway template here.]

There's a lot going on in the template, so we'll just give a "brief" overview of what's happening and point out some interesting syntax that you might use for your own projects.

CREATE AN EC2 INSTANCE AND S3 BUCKET

SFTP Gateway uses an EC2 server to upload files to S3. So we start off with an EC2 instance and S3 bucket:

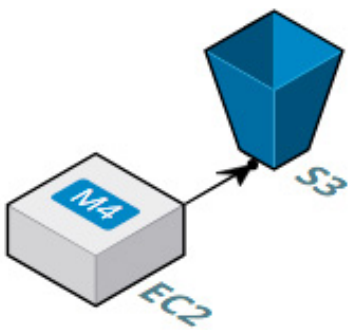
```
Resources:
  SFTPGatewayInstance:
    Type: AWS::EC2::Instance
    Metadata:
      AWS::CloudFormation::Init:
        config:
          commands:
            setup:
              command: /usr/local/bin/sftpgatewaysetup
    Properties:
      IamInstanceProfile: !Ref RootInstanceProfile
      ImageId: !FindInMap
        - RegionMap
        - !Ref AWS::Region
        - AMI
      InstanceType: !Ref EC2Type
      BlockDeviceMappings:
        - DeviceName: /dev/xvda
          Ebs:
            VolumeSize: !Ref DiskVolumeSize
            VolumeType: gp2
      KeyName: !Ref KeyPair
      SecurityGroupIds:
        - !Ref SFTPGatewaySG
      SubnetId: !Ref SubnetID
      Tags:
        - Key: Name
          Value: SFTPGateway Instance
        UserData:
          Fn::Base64: !Sub |
          #!/bin/bash
          yum update -y aws-cfn-bootstrap
          /opt/aws/bin/cfn-init --stack ${AWS::StackName}
--resource SFTPGatewayInstance --region ${AWS::Region}

  SFTPGatewayBucket:
    DeletionPolicy: Retain
    Type: AWS::S3::Bucket
    Properties:
      BucketName: !Sub sftpgateway-${SFTPGatewayInstance}
    DependsOn:
      - SFTPGatewayInstance
```

Here are a few things worth noting:

- The EC2 instance has a Metadata section in addition to its properties. We'll cover these in more detail below.
- The S3 bucket has a Deletion Policy of "Retain". This means you keep the S3 bucket if you delete the CloudFormation stack.
- The S3 BucketName uses an intrinsic function called "!Sub", which lets you do string interpolation. The syntax "${SFTPGatewayInstance}" gives you the EC2 instance ID, just like the "!Ref" function.

This is what we have so far:



Let's take a closer look at the EC2 instance metadata and properties:

```
SFTPGatewayInstance:
  Type: AWS::EC2::Instance
  Metadata:
    AWS::CloudFormation::Init:
      config:
        commands:
          setup:
            command: /usr/local/bin/sftpgatewaysetup
  Properties:
    IamInstanceProfile: !Ref RootInstanceProfile
    ImageId: !FindInMap
      - RegionMap
      - !Ref AWS::Region
      - AMI
    InstanceType: !Ref EC2Type
    BlockDeviceMappings:
      - DeviceName: /dev/xvda
```

```
        Ebs:
          VolumeSize: !Ref DiskVolumeSize
          VolumeType: gp2
      KeyName: !Ref KeyPair
      SecurityGroupIds:
        - !Ref SFTPGatewaySG
      SubnetId: !Ref SubnetID
      Tags:
        - Key: Name
          Value: SFTPGateway Instance
      UserData:
        Fn::Base64: !Sub |
          #!/bin/bash
          yum update -y aws-cfn-bootstrap
          /opt/aws/bin/cfn-init --stack ${AWS::StackName}
--resource SFTPGatewayInstance --region ${AWS::Region}
```

There's a lot going on here:

"CloudFormation::Init" – This is a powerful tool that lets you define config files and commands. In this case, we run a command called "sftpgatewaysetup" to initialize the software.

"ImageId" – This uses the "!FindInMap" intrinsic function that looks up an AMI ID based on the current region. The AMI mappings are located in the Mappings section of the CloudFormation

"BlockDeviceMappings" – This sets the disk drive type to solid state (gp2). It also points to a parameter named "DiskVolumeSize" which allows the user to define disk size at stack creation.

"KeyName" – This refers to an SSH key that you use to log into the server.

"UserData" – This lets you run bash commands on server launch. In this case, we use "cfn-init" to read the "CloudFormation::Init" metadata we defined earlier.

```
.
.
.
Parameters:
  EC2Type:
    Description: SFTPGateway Instance Type
    Type: String
    Default: t2.micro
    AllowedValues:
```

A lot of the properties above reference parameters. To the left is a snippet of the Parameters section of the template, which includes the "EC2Type", "DiskVolumeSize", and "KeyPair" parameters mentioned earlier:

```
      - t2.micro
      - t2.small
      - t2.medium
      - t2.large
      - m3.medium
      - m3.large
      - m3.xlarge
      - m4.large
      - m4.xlarge
      - c3.large
      - c3.xlarge
      - c4.large
      - c4.xlarge
      - r3.large
      - r3.xlarge
  KeyPair:
    Description: EC2 KeyPair
    Type: AWS::EC2::KeyPair::KeyName
    ConstraintDescription: Existing EC2 KeyPair.
  DiskVolumeSize:
    Default: 32
    Description: Disk volume size in GB. Must be at least 32.
    ConstraintDescription: Must be a number greater or equal to 32
    MinValue: 32
    Type: Number
```

Parameters let you pass dynamic values to make your template more flexible. Here are a few things to note:

• "AllowedValues" – This presents the user with a drop-down, so you don't have to worry about form validation.

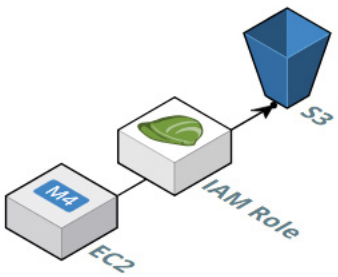
• "AWS::EC2::KeyPair::KeyName" – This is a special type that automatically presents the user with a list of key pairs in their
AWS account.

• "MinValue" – This provides form validation that gives an error if the user puts in a value that is too small.

CREATE AN IAM ROLE

In order for our EC2 instance to access S3, we need to grant it permissions using an IAM role.

The architecture looks like this:



The diagram shown above simplifies what's actually happening. If you look at the CloudFormation template, you'll see that there's more to it:

```
Resources:
.
.
.
  RootInstanceProfile:
    Type: AWS::IAM::InstanceProfile
    Properties:
      Path: /
      Roles:
        - !Ref S3WritableRole
  S3WritableRole:
    Type: AWS::IAM::Role
    Properties:
      AssumeRolePolicyDocument:
        Version: 2012-10-17
        Statement:
          - Effect: Allow
            Principal:
              Service: ec2.amazonaws.com
            Action: sts:AssumeRole
      Path: /
  RolePolicies:
    Type: AWS::IAM::Policy
    DependsOn:
      - SFTPGatewayInstance
    Properties:
      PolicyName: SFTPGatewayInstancePolicy
      PolicyDocument:
        Version: 2012-10-17
```

There are three resources involved when assigning permissions to an EC2 instance:

- "InstanceProfile" – You can't assign an IAM role directly to the EC2 instance, but you can assign an instance profile, which passes role information to the EC2 instance.
- "IAM::Role" – The EC2 instance can assume a role and inherit any permissions from the role, via the instance profile.
- "IAM::Policy" – This contains the actual permissions. The policy is associated with the role.

```
        Statement:
          - Effect: Allow
            Action: 's3:*'
            Resource: '*'
      Roles:
        - !Ref S3WritableRole
```

USING AN EXISTING PUBLIC SUBNET

The EC2 instance needs to be in a public subnet so that end users can access it via SFTP. This CloudFormation template doesn't create this public subnet. Rather, you select an existing subnet and pass it as a parameter to the template.

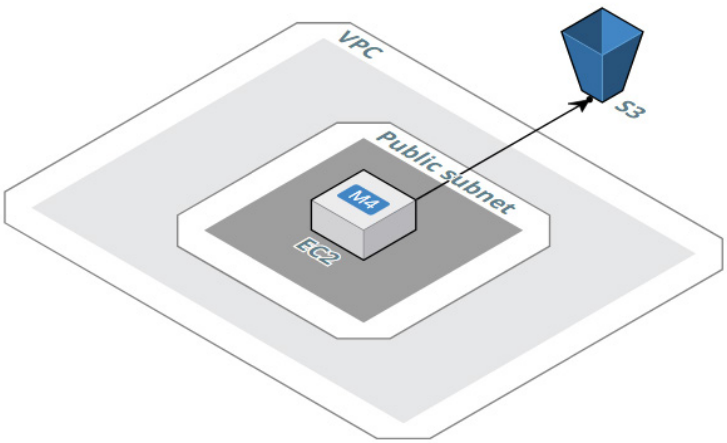
This happens here:

```
.
.
.
Parameters:
.
.
.
  VPCIdName:
    Description: Select the VPC to launch the SFTPGateway into
    Type: AWS::EC2::VPC::Id
  SubnetID:
    Description: Subnet ID
    Type: AWS::EC2::Subnet::Id
```

Here's what's going on:

- "AWS::EC2::Subnet::Id" – This is a special parameter type that lists existing subnets in your AWS account. The EC2 instance is provisioned in this subnet.
- "AWS::EC2::VPC::Id" – This is another special parameter type and it lists existing VPCs. In the next section, we will define a security group that gets provisioned in this VPC.

The architecture looks like this:



INCORPORATE SECURITY GROUP SETTINGS

In order for SFTP users to access the server, we use a Security Group to expose port 22 for specific IP addresses.

Here's the relevant code:

```
Resources:
.
.
.
  SFTPGatewaySG:
    Type: AWS::EC2::SecurityGroup
    Properties:
      GroupDescription: SFTPGateway Security Group
      VpcId: !Ref VPCIdName
      SecurityGroupIngress:
        - IpProtocol: tcp
          FromPort: 22
          ToPort: 22
          CidrIp: 0.0.0.0/0
```

The security group has a property called "SecurityGroupIngress", which accepts an array of rules. Here we have a single rule that allows all traffic (0.0.0.0/0) on TCP port 22.

ADDING AN ELASTIC IP

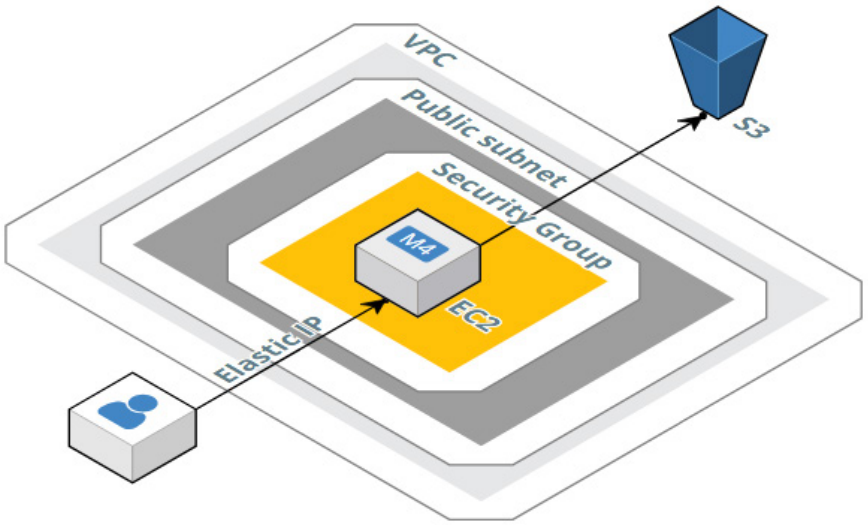
Finally, we need to create a static IP so that our public IP address doesn't change each time the server shuts down.

```
Resources:
.
.
.
  IPAddress:
    Properties:
      Domain: vpc
      InstanceId: !Ref SFTPGatewayInstance
    Type: AWS::EC2::EIP
```

Here's a brief explanation:

- "Domain" – This is set to "vpc", since we're using VPC instead of classic networking.
- "InstanceId" – This is the instance ID of the EC2 server that receives the IP address

We wind up with this final wonderful architecture:



OUTPUTS SECTION

The Outputs section lets you display concise information for easy access. Here, we show the public IP address to make it easier to connect for the first time.

```
Outputs:
  ElasticIP:
    Value: !Ref IPAddress
    Description: Elastic IP address
```

CONCLUSION

Now you can easily and securely upload your files to Amazon S3 via SFTP! Give the product a try by visiting the SFTP Gateway AWS Marketplace page.

You've also learned how to incorporate EC2, IAM, S3, Security Groups, and more to facilitate this file transfer.

Now let's learn how to use CloudFormation to launch a Redshift stack!

HOW TO CREATE A REDSHIFT STACK WITH AWS CLOUDFORMATION

USE THIS CLOUDFORMATION TEMPLATE TO LAUNCH REDSHIFT INTO YOUR VPC SUBNET WITH S3 AS THE DATA SOURCE

Our third and final template creates an Amazon Redshift stack. Redshift is a data warehousing solution that allows you to run complex data queries on huge data sets within seconds (it's pretty awesome).

You can use it to generate reports and analyze customer data. This stack will help you get up and running with Redshift.

There are a number of ways to get your data into Redshift. In this template, we use S3 as the data source.

For simplicity, we'll put Redshift in a VPC subnet so that you can connect directly to it without setting up a VPN or proxy (note: we don't recommend this for production environments).

For some baseline security, Redshift will be locked down to your specific IP address.

Here's the entire Redshift template:

```
AWSTemplateFormatVersion: 2010-09-09
Description: Redshift Stack
Conditions:
  SingleNode: !Equals [ !Ref RedshiftNodeCount, 1 ]
Parameters:
  SubnetA:
    Type: String
    Type: AWS::EC2::Subnet::Id
    Description: Make sure this belongs to the VPC specified below
    (e.g. 172.31.0.0/20)
  SubnetB:
    Type: String
    Type: AWS::EC2::Subnet::Id
    Description: Make sure this is different from the subnet above
    (e.g. 172.31.16.0/20)
  VPCID:
    Type: String
    Type: AWS::EC2::VPC::Id
    Description: Select a VPC (e.g. 172.31.0.0/16)
  DataBucketName:
    Type: String
    Description: S3 data bucket name
  DatabaseName:
    Type: String
    Description: Database name
  MasterUsername:
    Type: String
    Description: Master user name for Redshift
    Default: admin
  MasterUserPassword:
    Type: String
    Description: Master password for Redshift
    (used mixed case and numbers)
    NoEcho: true
  DeveloperIPAddress:
    Type: String
    Description: Your public IP address (see http://checkip.dyndns.org/)
  RedshiftNodeCount:
    Type: Number
    Description: Number of Redshift nodes
```

```
    Default: 1
    MinValue: 1
    ConstraintDescription: Must be a number greater or equal to 1
Resources:
  RedshiftCluster:
    Type: AWS::Redshift::Cluster
    Properties:
      ClusterSubnetGroupName: !Ref RedshiftClusterSubnetGroup
      ClusterType: !If [ SingleNode, single-node, multi-node ]
      NumberOfNodes: !If [ SingleNode, !Ref ‚AWS::NoValue‘, !Ref
RedshiftNodeCount ] #‘

      DBName: !Sub ${DatabaseName}
      IamRoles:
        - !GetAtt RawDataBucketAccessRole.Arn
      MasterUserPassword: !Ref MasterUserPassword
      MasterUsername: !Ref MasterUsername
      PubliclyAccessible: true
      NodeType: dc1.large
      Port: 5439
      VpcSecurityGroupIds:
        - !Sub ${RedshiftSecurityGroup}
      PreferredMaintenanceWindow: Sun:09:15-Sun:09:45
  DataBucket:
    Type: AWS::S3::Bucket
    Properties:
      BucketName: !Sub ${DataBucketName}
  RawDataBucketAccessRole:
    Type: AWS::IAM::Role
    Properties:
      AssumeRolePolicyDocument:
        Version: 2012-10-17
        Statement:
          -
            Effect: Allow
            Principal:
              Service:
                - redshift.amazonaws.com
            Action:
              - sts:AssumeRole
  RawDataBucketRolePolicy:
    Type: AWS::IAM::Policy
```

```
  Properties:
    PolicyName: RawDataBucketRolePolicy
    PolicyDocument:
      Version: 2012-10-17
      Statement:
        -
          Effect: Allow
          Action: s3:ListAllMyBuckets
          Resource: arn:aws:s3:::*
        -
          Effect: Allow
          Action:
            - 's3:Get*'
            - 's3:List*'
          Resource: '*'
        -
          Effect: Allow
          Action: cloudwatch:*
          Resource: "*"
    Roles:
      - !Ref RawDataBucketAccessRole
RedshiftClusterSubnetGroup:
  Type: AWS::Redshift::ClusterSubnetGroup
  Properties:
    Description: Cluster subnet group
    SubnetIds:
      - !Ref SubnetA
      - !Ref SubnetB
RedshiftSecurityGroup:
  Type: AWS::EC2::SecurityGroup
  Properties:
    GroupDescription: Enable JDBC port
    VpcId: !Ref VPCID
    SecurityGroupIngress:
      -
        CidrIp: !Sub ${DeveloperIPAddress}/32
        FromPort: 5439
        ToPort: 5439
        IpProtocol: tcp
        Description: IP address for your dev machine
      -
```

```
          SourceSecurityGroupId: !Ref AccessToRedshiftSecurityGroup
          FromPort: 5439
          ToPort: 5439
          IpProtocol: tcp
          Description: Access to redshift
  AccessToRedshiftSecurityGroup:
    Type: AWS::EC2::SecurityGroup
    Properties:
      GroupDescription: Access to Redshift access
      VpcId: !Ref VPCID
  InternalSecurityGroupIngress:
    Type: AWS::EC2::SecurityGroupIngress
    Properties:
      IpProtocol: tcp
      FromPort: 0
      ToPort: 65535
      SourceSecurityGroupId: !Ref RedshiftSecurityGroup
      GroupId: !Ref RedshiftSecurityGroup
Outputs:
  RedshiftClusterEndpointAddress:
    Description: Redshift Cluster Endpoint Address
    Value: !GetAtt RedshiftCluster.Endpoint.Address
  RedshiftClusterEndpoint:
    Description: Redshift Cluster Endpoint
    Value:
      Fn::Join:
        - ""
        - - 'jdbc:redshift://'
          - !GetAtt RedshiftCluster.Endpoint.Address
          - ':5439/'
          - !Sub ${DatabaseName}
```

You can download this CloudFormation template by clicking here.

Let's explore what's going on here.

CREATION OF THE REDSHIFT CLUSTER

The first thing we do is create the Redshift cluster.

Please note that the code snippet below is simplified for demonstration purposes and doesn't yet match the code we provided in the overall template above. We'll revisit and explain the additional Redshift properties in a later section.

```
.
.
.
Resources
  RedshiftCluster:
    Type: AWS::Redshift::Cluster
    Properties:
      ClusterType: SingleNode
      NumberOfNodes: 1
      DBName: !Sub ${DatabaseName}
      MasterUserPassword: !Ref MasterUserPassword
      MasterUsername: !Ref MasterUsername
      PubliclyAccessible: true
      NodeType: dc1.large
```

Here are the key aspects of this code:

- "ClusterType": This can be "SingleNode" or "MultiNode". For now, we hard-code "SingleNode".
- "NumberOfNodes": Since we're using "SingleNode", this has to be set to 1.
- "DBName": This refers to a parameter in the Parameters section called "DatabaseName", which becomes the name of our Redshift database.
- "MasterUsername": This is another parameter that sets the master user name.
- "MasterUserPassword": This is also a parameter for setting the master password.
- "PubliclyAccessible": This is set to true so that you can connect to it easily.
- "NodeType": "dc1.large" is the least expensive node type.

Like we mentioned prior, there are a few more Redshift properties that we've included in our overall template that we'll explain in a later section titled "More Redshift cluster properties".

SET UP S3 AS A DATA SOURCE

Redshift can load data from different data sources. In this example, we'll be using S3.

To set this up, we have to create an S3 bucket and an IAM role that grants Redshift access to S3. This is what the code looks like:

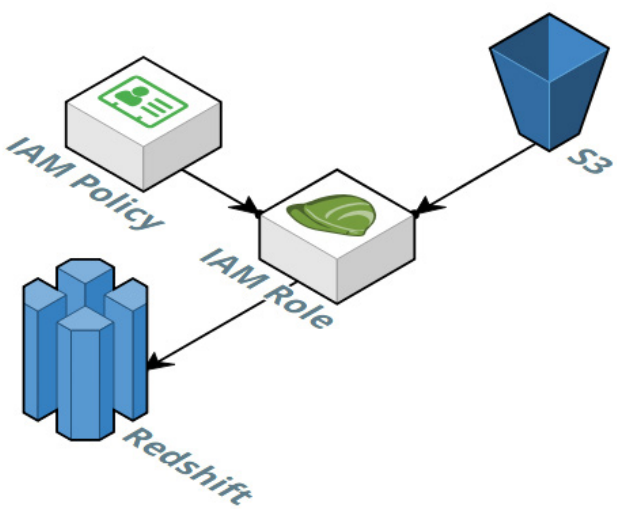
```
Resources:
.
.
.
  DataBucket:
    Type: AWS::S3::Bucket
    Properties:
      BucketName: !Sub ${DataBucketName}
  RawDataBucketAccessRole:
    Type: AWS::IAM::Role
    Properties:
      AssumeRolePolicyDocument:
        Version: 2012-10-17
        Statement:
          -
            Effect: Allow
            Principal:
              Service:
                - redshift.amazonaws.com
            Action:
              - sts:AssumeRole
  RawDataBucketRolePolicy:
    Type: AWS::IAM::Policy
    Properties:
      PolicyName: RawDataBucketRolePolicy
      PolicyDocument:
        Version: 2012-10-17
        Statement:
```

```
            -
              Effect: Allow
              Action: s3:ListAllMyBuckets
              Resource: arn:aws:s3:::*
            -
              Effect: Allow
              Action:
                - 's3:Get*'
                - 's3:List*'
              Resource: '*'
            -
              Effect: Allow
              Action: cloudwatch:*
              Resource: "*"
        Roles:
          - !Ref RawDataBucketAccessRole
```

Here's a quick overview of what's going on:

- "S3::Bucket": The bucket name comes from a parameter called "DataBucketName".
- "IAM::Role": This is the IAM role that allows access to S3. It doesn't have any permissions yet but it allows the Redshift service to assume this role.
- "IAM::Policy": This contains a list of permissions for accessing S3 and Cloudwatch. The policy associates itself with the IAM Role.

So far, the architecture looks like this:



CREATE VPC AND PUBLIC SUBNETS

You usually want to put databases in a private subnet, like we mentioned in our VPC template chapter. But in the early stages of a project, you might want direct access to Redshift from your development machine.

We don't recommend this for production environments, but in this development case, you can start off by putting Redshift into your VPC subnet.

```
Resources:
.
.
.
  RedshiftClusterSubnetGroup:
    Type: AWS::Redshift::ClusterSubnetGroup
    Properties:
      Description: Cluster subnet group
      SubnetIds:
        - !Ref SubnetA
        - !Ref SubnetB
```

We can't put Redshift in a subnet directly, so here we put Redshift in something called a "ClusterSubnetGroup". You can then add multiple subnets to the "ClusterSubnetGroup". These subnets should be in different availability zones, which helps with high availability.

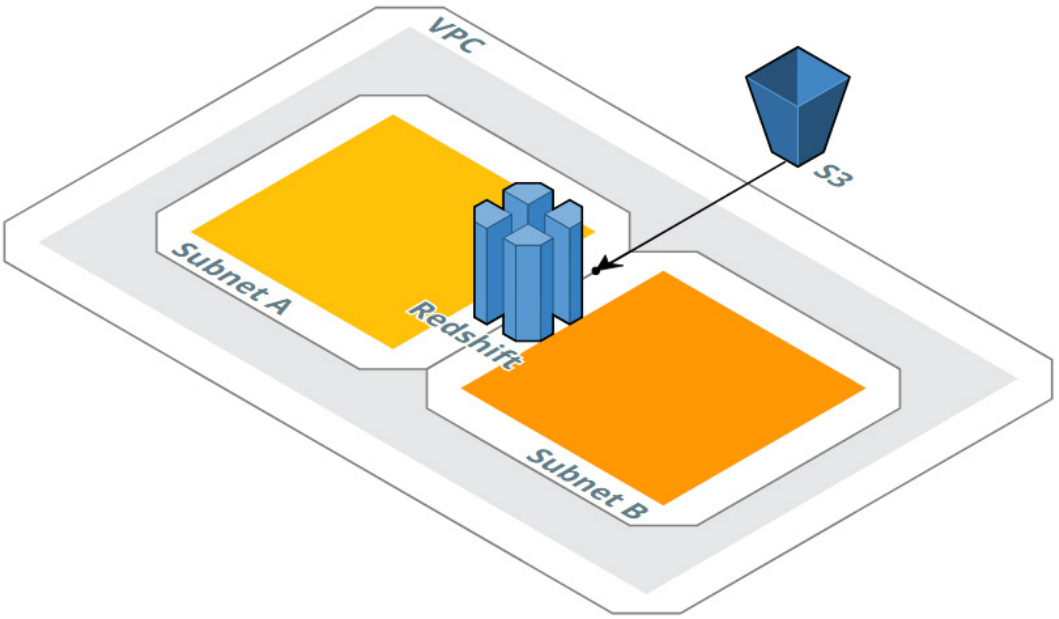
SUBNET AND VPC PARAMETERS

The Redshift CloudFormation template doesn't create any subnets or networks of its own. Instead, it asks you for parameters — two public subnets and a VPC.

```
Parameters:
  SubnetA:
    Type: String
    Type: AWS::EC2::Subnet::Id
    Description: Make sure this belongs to the VPC specified
below (e.g. 172.31.0.0/20)
  SubnetB:
    Type: String
    Type: AWS::EC2::Subnet::Id
    Description: Make sure this is different from the subnet
above (e.g. 172.31.16.0/20)
  VPCID:
    Type: String
    Type: AWS::EC2::VPC::Id
    Description: Select a VPC (e.g. 172.31.0.0/16)
```

You can just pick the VPC and public subnets that come by default in every region of each AWS account.

We wind up with this architecture:



CREATE A SECURITY GROUP

So far, the Redshift cluster is in a public subnet. But before we can connect to it, we have to add a security group to allow port traffic to Redshift.

```
Resources:
.
.
.
  RedshiftSecurityGroup:
    Type: AWS::EC2::SecurityGroup
    Properties:
      GroupDescription: Enable JDBC port
      VpcId: !Ref VPCID
      SecurityGroupIngress:
        -
          CidrIp: !Sub ${DeveloperIPAddress}/32
          FromPort: 5439
          ToPort: 5439
          IpProtocol: tcp
          Description: IP address for your dev machine
```

This allows port 5439 traffic, which is the default TCP port for Redshift. This is locked down to the public IP address of your computer, which you provide via the CloudFormation parameter "DeveloperIPAddress".

CONFIGURE SECURITY GROUP ACCESS

During development, you'll want to access Redshift directly from your development machine. But eventually, you want to make calls to Redshift from an application, such as AWS Lambda.

For this, you need to create other security groups and grant these access to Redshift.

```
Resources:
.
.
.
  RedshiftSecurityGroup:
    Type: AWS::EC2::SecurityGroup
    Properties:
      GroupDescription: Enable JDBC port
      VpcId: !Ref VPCID
      SecurityGroupIngress:
        -
          CidrIp: !Sub ${DeveloperIPAddress}/32
          FromPort: 5439
          ToPort: 5439
          IpProtocol: tcp
          Description: IP address for your dev machine
        -
          SourceSecurityGroupId: !Ref AccessToRedshiftSecurityGroup
          FromPort: 5439
          ToPort: 5439
          IpProtocol: tcp
          Description: Access to redshift
  AccessToRedshiftSecurityGroup:
    Type: AWS::EC2::SecurityGroup
    Properties:
      GroupDescription: Access to Redshift access
```

```
      VpcId: !Ref VPCID
  InternalSecurityGroupIngress:
    Type: AWS::EC2::SecurityGroupIngress
    Properties:
      IpProtocol: tcp
      FromPort: 0
      ToPort: 65535
      SourceSecurityGroupId: !Ref RedshiftSecurityGroup
      GroupId: !Ref RedshiftSecurityGroup
```

This example builds off of the "RedshiftSecurityGroup" from the previous section. Here, we're configuring two types of access:

- "AccessToRedshiftSecurityGroup": This is an additional security group that you might assign to an application, such as AWS Lambda. We add a security group ingress rule that allows inbound traffic on port 5439.
- "InternalSecurityGroupIngress": This is a standalone rule that allows resources in one "RedshiftSecurityGroup" to access another. It's configured as a standalone ingress rule, because CloudFormation resources can't reference themselves within their own properties.

MORE REDSHIFT CLUSTER PROPERTIES

As you wrap up development, you'll want to start thinking about deploying to production. Here are a few tweaks to the Redshift cluster that we created in the first section that might come in handy:

```
.
.
.
Conditions:
  SingleNode: !Equals [ !Ref RedshiftNodeCount, 1 ]
Parameters:
```

```
.
.
.
  RedshiftNodeCount:
    Type: Number
    Description: Number of Redshift nodes
    Default: 1
    MinValue: 1
    ConstraintDescription: Must be a number greater or equal to 1
.
.
.
Resources:
.
.
.
  RedshiftCluster:
    Type: AWS::Redshift::Cluster
    Properties:
      ClusterSubnetGroupName: !Ref RedshiftClusterSubnetGroup
      ClusterType: !If [ SingleNode, single-node, multi-node ]
      NumberOfNodes: !If [ SingleNode, !Ref 'AWS::NoValue', !Ref
RedshiftNodeCount ]
      DBName: !Sub ${DatabaseName}
      IamRoles:
        - !GetAtt RawDataBucketAccessRole.Arn
      MasterUserPassword: !Ref MasterUserPassword
      MasterUsername: !Ref MasterUsername
      PubliclyAccessible: true
      NodeType: dc1.large
      Port: 5439
      VpcSecurityGroupIds:
        - !Sub ${RedshiftSecurityGroup}
      PreferredMaintenanceWindow: Sun:09:15-Sun:09:45
```

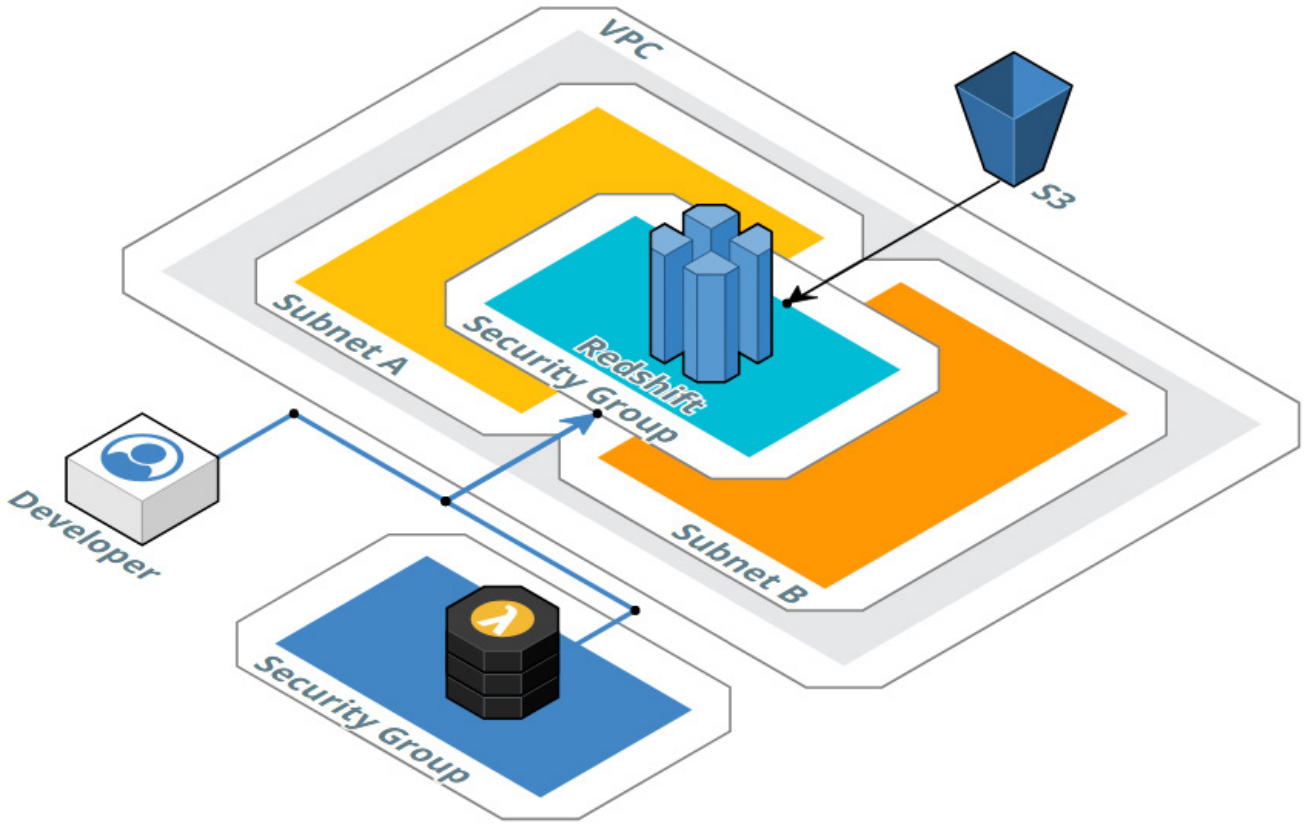
We add a parameter called "RedshiftNodeCount". This represents how many Redshift nodes you want in your cluster.

We use a condition called “SingleNode” that checks if we have just one node. If so, we pass “single-node” to the “ClusterType” property. Otherwise, we pass in “multi-node” if more than one node was specified.

The “NumberOfNodes” property gets a little tricky. If there are multiple nodes, we can just pass in the “RedshiftNodeCount”. But if there’s just a single node, we get an error if we populate “NumberOfNodes” with any information, even if it’s just the number “1”.

The way around this is to use the pseudo parameter “AWS::NoValue”. If there’s just a single node, we pass “AWS::NoValue” to “NumberOfNodes” (which has the same effect as deleting that property).

Here’s the final architecture:



In this diagram, you can access your Redshift cluster from both your development machine, and an application such as AWS Lambda.

OUTPUTS

Once you spin up a Redshift cluster, the first thing you want to do is connect to it. One useful piece of information to output would be the Redshift cluster endpoint.

```
.
.
.
Outputs:
  RedshiftClusterEndpointAddress:
    Description: Redshift Cluster Endpoint Address
    Value: !GetAtt RedshiftCluster.Endpoint.Address
  RedshiftClusterEndpoint:
    Description: Redshift Cluster Endpoint
    Value:
      Fn::Join:
        - ""
        - - 'jdbc:redshift://'
        - !GetAtt RedshiftCluster.Endpoint.Address
        - ':5439/'
        - !Sub ${DatabaseName}
```

Here we have the "RedshiftClusterEndpointAddress", which gives you the DNS hostname of the Redshift cluster. To make things even more convenient, we construct a JDBC url in the format of:

jdbc:redshift://examplecluster.cg034hpkmmjt.us-east-1.redshift.amazonaws.com:5439/dbname

which you can paste into your database client software.

CONCLUSION

Redshift is a really powerful data warehousing tool that makes it fast and simple to analyze your data and glean insights that can help your business. This CloudFormation template will help you automate the deployment of and get you going with Redshift.

Overall, there's so much that you can do with CloudFormation and it's difficult to review every little detail. But we hope that walking through these templates gives you a better idea of the power of CloudFormation and how you can use it to manage your AWS deployments.

NEED MORE INFORMATION?

To learn more about how Thorn Technologies can help you incorporate Infrastructure as Code, please contact us at info@thorntech.com or visit www.thorntech.com.