AWS POWERSHELL

Deploy
Manage
Automate

BY BRYCE MCDONALD

AWS PowerShell

Deploy, Manage, Automate

Bryce McDonald

This book is for sale at http://leanpub.com/awspowershell

This version was published on 2019-11-06

Tweet This Book!

Please help Bryce McDonald by spreading the word about this book on Twitter!

The suggested tweet for this book is:

I'm becoming a master of AWS and PowerShell!

The suggested hashtag for this book is #AWSPowerShell.

Find out what other people are saying about the book by clicking on this link to search for this hashtag on Twitter:

#AWSPowerShell

To my wife and my boy

may I always live up to your expectations

Contents

Introduction **1**
Getting the DevOps Mentality 1
Don’t be a “Left Click Engineer” 2
What Makes a Good Automator? 3
Why PowerShell over the AWS CLI? 4
AWS and You: The Shared Responsibility Model 4
Code Samples 5

Chapter 1: Getting Set Up with AWS and PowerShell **6**
Creating your AWS Account 6
Create an IAM User and Generate Access Keys 11
Installing AWS PowerShell for Windows 13
Installing AWS PowerShell for Linux/MacOS 15
Importing the Module 15
Configuring AWS PowerShell 16
AWS Account Best Practices 17
Summary 21

Chapter 2: Getting Started using PowerShell on AWS **22**
Traditional Cmdlet Discovery 22
Built-In Discovery Cmdlet 25
Pipelining Commands 29

Chapter 3: Authentication **30**
Credential Management 30
Configuring Federated Identity Providers 31

Chapter 4: The big 3, IAM, EC2, S3 . . . . **32**
Managing IAM with PowerShell 32
Managing EC2 with PowerShell 34
Managing S3 with PowerShell 34

Chapter 5: Integrating more AWS Services **35**
Compute Resources 35
Storage Resources 35

DataBase Resources 35
Migration Resources 35
Management Tools 35
Machine Learning 35
Application Integration 35
Analytics 35

Chapter 6: AWS Best Practices 36
Access Management 36
Reliability 36
Performance 37
Cost 37
Account Hygiene 37
Taking Action to Remediate 37
Amazon Well-Architected Reviews 38

Chapter 7: Security in AWS 39
Built-in Mitigations for Common Attacks 39
Compliance Program 39

Chapter 8: Billing 40
How to manage budgets 40
How to monitor your spend 40
How to reduce your bill 40

Introduction

This book is going to make a couple of assumptions about you.

First, I already know you're an extremly smart person because you bought this book and are taking the time to better yourself. There's nothing more crucial in career development than to spend a little time learning a skill, sharpening your toolset, or even just exploring alternative solutions to something you're already an expert in. Because I already know this about you, I know we're going to have a good time together.

Second, I'm going to assume that you know something about PowerShell. There are many great books and resources to learn this such as Don Jones "Learn PowerShell in a Month of Lunches", resources like Adam Bertram's PluralSight series on PowerShell toolmaking, and the recently launched TechSnips[1] platform which gives you tasked based and laser focused short videos on specific PowerShell topics. I'll do my best to cover some of the more advanced concepts of PowerShell, but if you're confused by what the terms `Get-Help` and `Get-Command` mean, or if you've never looked at or written a PowerShell function, I'd ask that you go and bolster your skills a little more before coming back to this book.

Last, I'm going to assume that you have access to any kind of AWS environment (free tier stuff is fine, and I'll make my examples as free-tier friendly as possible). Following along with my code samples and screenshots will do you a bit of good, but without getting your hands into a real AWS environment, you won't be able to fully grasp the concepts that I'm going to lay out. Once you see a few services all ticking away together, AWS will begin to conceptually make a lot more sense, especially if you can toy with the individual services and tweak the performance in a way that only you can do.

Before we get started, I'm going to pump the brakes for a second and talk philosophy rather than technology. I want to make sure that you have an understanding of **why** we do the things we do with PowerShell and with AWS. Once you have the foundation of DevOps principles, and once you have the itch to automate everything, you'll be in the right head space for the rest of this book.

Now that I've given you the appropriate disclaimer, let's proceed. I'll help you get into the DevOps mindset, then we'll talk about the demise of the "Left Click Engineer", followed by the characteristics of a good automator, why I choose PowerShell for the job, then wrap it up with some info about where to find the code set used in this book.

Getting the DevOps Mentality

I hate buzzwords. I really do. Sometimes, however, they're necessary. This is the case for DevOps. I'm not going to tell you how it's going to synergize your optimal output efficiency or how it's going

[1]https://www.techsnips.io

to help you win the day, but I *am* going to talk about how DevOps is a necessity in any modern agile software development or operations management team. DevOps is a blend of development and operations which allows each team to work together, rather than having siloed units of work that meet once a week in a “scrum”.

There are a few defining principles of DevOps that I’d like to impress upon you.

Servers are Cattle, not Pets

You may have heard this before, but it’s very much worth repeating. Server management used to be a very intimate process. About once a week or more, I’d drive down to the datacenter, swipe my access key past whatever security personell was present that day, go to *my* rack in *my* corner in the datacenter, and I’d lovingly dust off the monitor and keyboard before performing a few management tasks. By hand, I would replace hard drives, and by hand, I would update, restart, configure, and deploy new services. My servers were my babies, and I’d do everything I could to protect them, play with them, and extend their life. It was always a sad day when one of my servers died, because at that point I felt as though I was failing my job as a system administrator.

Today, I treat my managed servers in a much more callous fashion. It’s not working? Kill it. Redeploy. Spin up a new one. Anymore I won’t waste my time trying to troubleshoot an issue on one machine, since it is so much faster to spin up a known working configuration than it is to go back and figure out what’s wrong. Don’t take this to the extreme, though. There are times where troubleshooting is necessary and failure is unexpected, but in my ideal environment I don’t have to go back and spend hours upon hours trying to get my service back online. I can confidently deploy my systems and safely expect them to be available to my users.

How can I be so confident? I’m glad you asked.

Fail Like You Mean It

With DevOps, the motto is pretty simple. Fail fast, fail early, fail often, but learn from those mistakes and don’t make them again. Your developer project workflow should be able to account for failures, you should have practiced rollbacks, and you should have steps in place to be able to minimize service downtime while maximizing code throughput. Code failures shouldn’t be pushished, full stop. Code failures provide an opportunity to learn about your service and what breaks it, and it also reveals the necessity of testing your code before it gets deployed. A coding failure doesn’t mean someone messed up, it means that we have more work to do. Collectively, we should be able to identify what went wrong, figure out where we need to fix the code, and how we should write our unit and integration tests to make sure that the failure doesn’t happen again.

Don’t be a “Left Click Engineer”

There’s a concept that we’ve been talking recently at my local Kansas City PowerShell User Group. As Microsoft Engineers, a lot of us are surrounded by what we’ve dubbed “Left Click Engineers”.

These are the folks who have to have a GUI to be able to perform anything meaningful in a work environment. There was a time and a place in Microsoft's architecture where work like this was rewarded. If you look at GPO's, (some) Active Directory snap-ins, the Exchange Management Console, and a few other GUI's (server manager, for instance), there were many places where it was simpler to use the GUI than it was to do anything else.

Let's gain a little perspective on the "new Microsoft" real quick. Look at how the climate at Microsoft is moving. There are API's for everything, including infrastructure in the cloud. With the acquisition of GitHub, and the rumored acquisition of RedHat, there are going to be fewer and fewer opportunities to guarantee that your GUI is going to exist on the right architecture. I haven't even gotten to whether or not your GUI of choice is going to scale. There's no reason you should be bound to a GUI. When was the last time you took a look at Windows Server 2016. Did you remember to install the desktop experience on it? Did you even deploy it by hand, or just request the resource from AWS/Azure? Trust me, if you're not automating, then you are the one being automated.

What Makes a Good Automator?

I will always insit that good automators are really lazy at heart. I think good automators are *so* lazy that they're willing to do a bunch of work to ensure that they never have to work again. In my case, I can't stand to do any manual tasks. I already mentioned how the GUI is going away, but I want command line input to go away as well. In fact, I don't even want to have to type my commands, I want them to run on schedules or respond to triggers. I always think of what I could do to eliminate button presses and key strokes to the point where I don't have to do them at all anymore. In a previous organizational structure, I was the ranking new guy at a large software developer. I was given task after task, and told the schedule they were to run on, the code that was supposed to run, and how to manually log on to whatever server and perform that task. To me, that wasn't going to fly. Within a few weeks I had the SQL queries going, the reports run, and the files being dropped into the right network share and a message in slack telling me it was all done. My time sheets looked pretty abysmal after that, to be honest.

Speaking of time sheets, I hate them, but I do find one thing redeemable about them. It becomes very easy to identify where you're spending all your time. At the aforementioned software company I worked for, I would do exactly this with my time sheet. I'd take the line item that I spent the most amount of time on, and figure out a way to not have to spend that much time on it ever again. Soon, instead of spending 20 hours a week on this task, and 20 hours a week on the rest of my job, I had 20 hours a week of free time. The free time allowed me to take on more responsibility, and I found I was spending 10 hours on a new task, with 30 hours for the rest of my job. The next week, I had 10 hours of free time because I automated that task… You see where I'm going with this? You'll never run out of work to do if you use your timesheet to figure out ways to continue to automate. You'll naturally be given more responsibliity (or in the case of a business owner or manager, you'll be able to go out and look for new opportunities) in order to fill the gaps.

Why PowerShell over the AWS CLI?

Both the AWS CLI and AWS Tools for PowerShell are built on top of the AWS SDK. The CLI is based on the Python SDK, and the PowerShell module is built on top of the .NET SDK. Both allow you to script your operations to manage your resources, and both (as of PowerShell Core) are cross platform. For all sakes and purposes, they are quite similar in how they work. If you're a python native, then you should be very comfortable with the AWS CLI. For PowerShell natives, you'll find that this module is just as intuitive and easy to use as the rest of PowerShell is.

In my opinion (and yeah, it's pretty biased), PowerShell is the best scripting language to perform operational tasks. As a high level language, you can easily and quickly create powerful scripts to manage entire environments. If some functionality is missing that you want, module creation is incredibly easy and simple, plus you can package and distribute your module natively with other Windows tools. Last, if you can't do something with PowerShell, then I am willing to bet that there's a .NET method you could call to perform the task, and you can embed that within your PowerShell scripts. By combining PowerShell with AWS, you can manage environments at scale, and I'll show you how to do that throughout the rest of this book.

AWS and You: The Shared Responsibility Model

In AWS, there's a concept called the "Shared Responsibility Model" in which AWS takes responsibiliity for the cloud, while responsibility for the components we place in the cloud is passed on to you and I. The folks at AWS take security very seriously, so the fact that they pitch in to secure the cloud in general is a good step toward having a very mature security posture. Your milage may vary depending on the service you're using, but it's a general rule of thumb that if you configured it, then you'll have to take care of it. For instance, AWS will take care of the following components:

- AWS Regions
- Availability Zones
- Edge Locations
- and more.

Whereas you and I are going to be responsible for:

- Data placed in S3
- Identity and Access Management (Security Groups)
- Server-side encryption
- and more.

AWS Manages the behind the scenes infrastructure so all we have to do is worry about managing our code and our environment. Because PowerShell is an automation powerhouse, it's a great blend of technologies to have these two coming together. Our scripts can be simpler despite getting more done on AWS.

Code Samples

At this point, I’m sure you’re wanting to know where the code is! All of the code in this book can be found on my personal GitHub[2]. I’ve created it using the MIT license, so if you decide to use it just give me a shout out on twitter or in the credits (but don’t send your lawyer after me).

Without further adieu, let’s get started.

[2]https://github.com/mcdonaldbm/AWS_PowerShell_Book

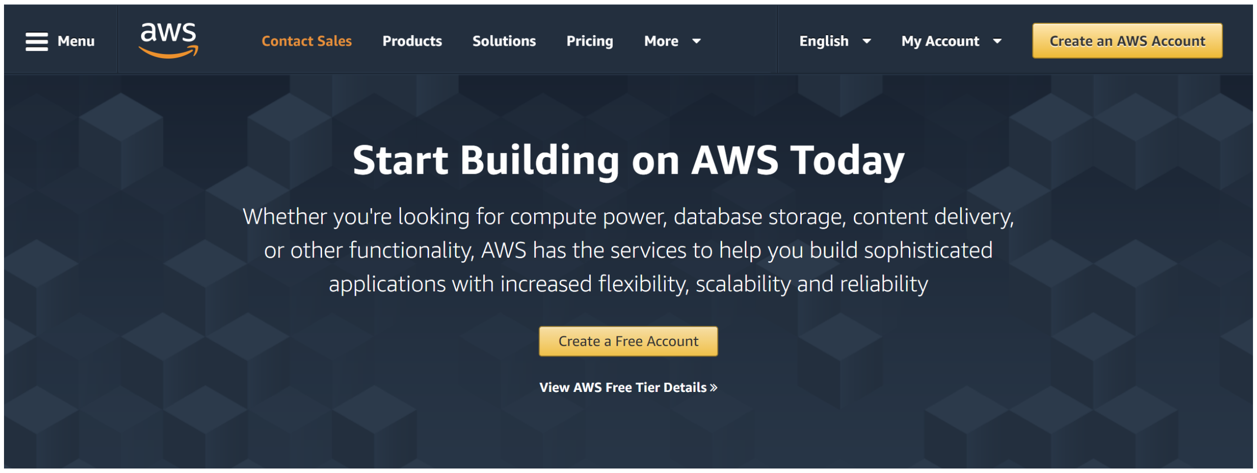
Chapter 1: Getting Set Up with AWS and PowerShell

As mentioned in the introduction, the AWS Tools for PowerShell and AWS Tools for PowerShell Core are modules that are written on top of the .NET SDK for Amazon Web Services. Because it's a PowerShell module, we get all of the PowerShell-y goodness that we're used to, like its ability to manipulate objects with ease, treating adminstrators as first class users, and its robust pipelining capabilities.

Before we can get started with PowerShell on AWS, we need to start with an AWS account and some access keys. Then, we'll install and configure the AWS Powershell tools for your version of Windows, iOS, or Linux. We'll go deeper into authentication options later on, so if you're looking to configure multiple profiles you may want to skip ahead a bit. Finally, we'll dive in to PowerShell to start locking down your account and setting some best practices in the settings for new accounts.

Creating your AWS Account

Obviously, we'll need an AWS account to get started. If you don't have one yet, you can follow along to sign up for one.

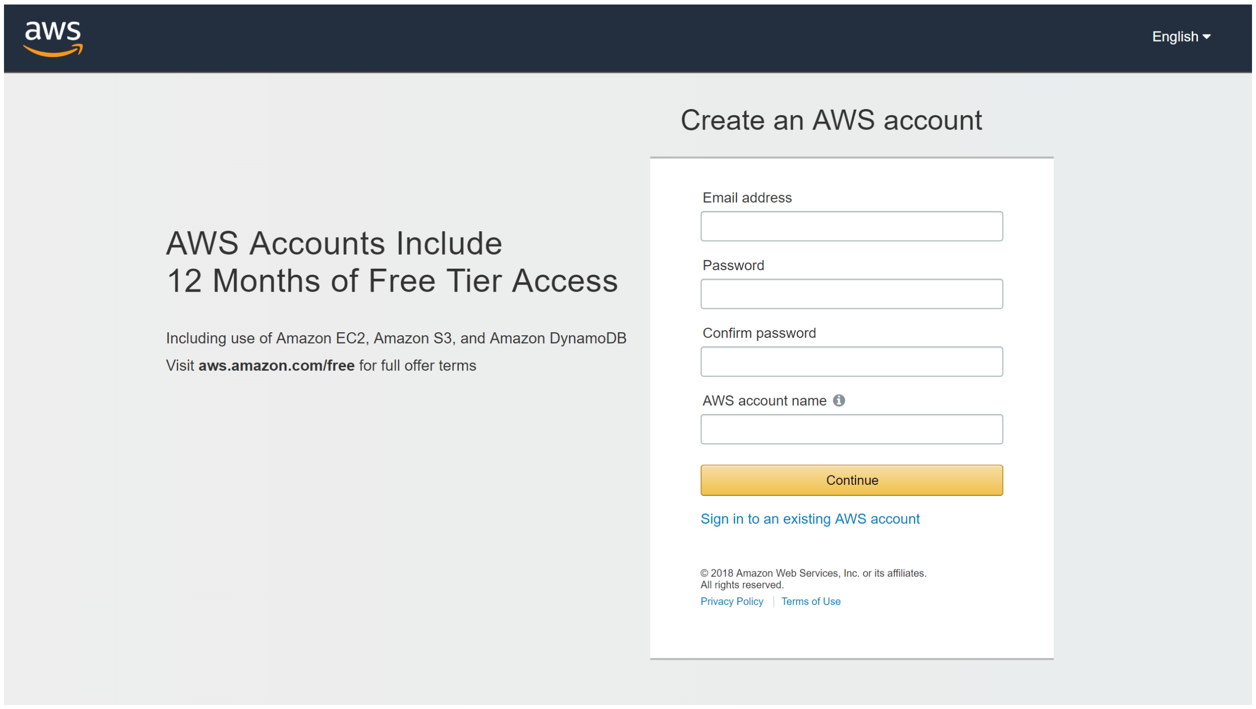


Whether you're looking for compute power, database storage, content delivery, or other functionality, AWS has the services to help you build sophisticated applications with increased flexibility, scalability and reliability

Sign up for AWS

We'll first start by visiting the AWS base page.[3]. From here, you'll select "Create Account" and you'll be asked for some basic information.

[3]https://aws.amazon.com



Create an AWS account

Email address

Password

Confirm password

AWS account name

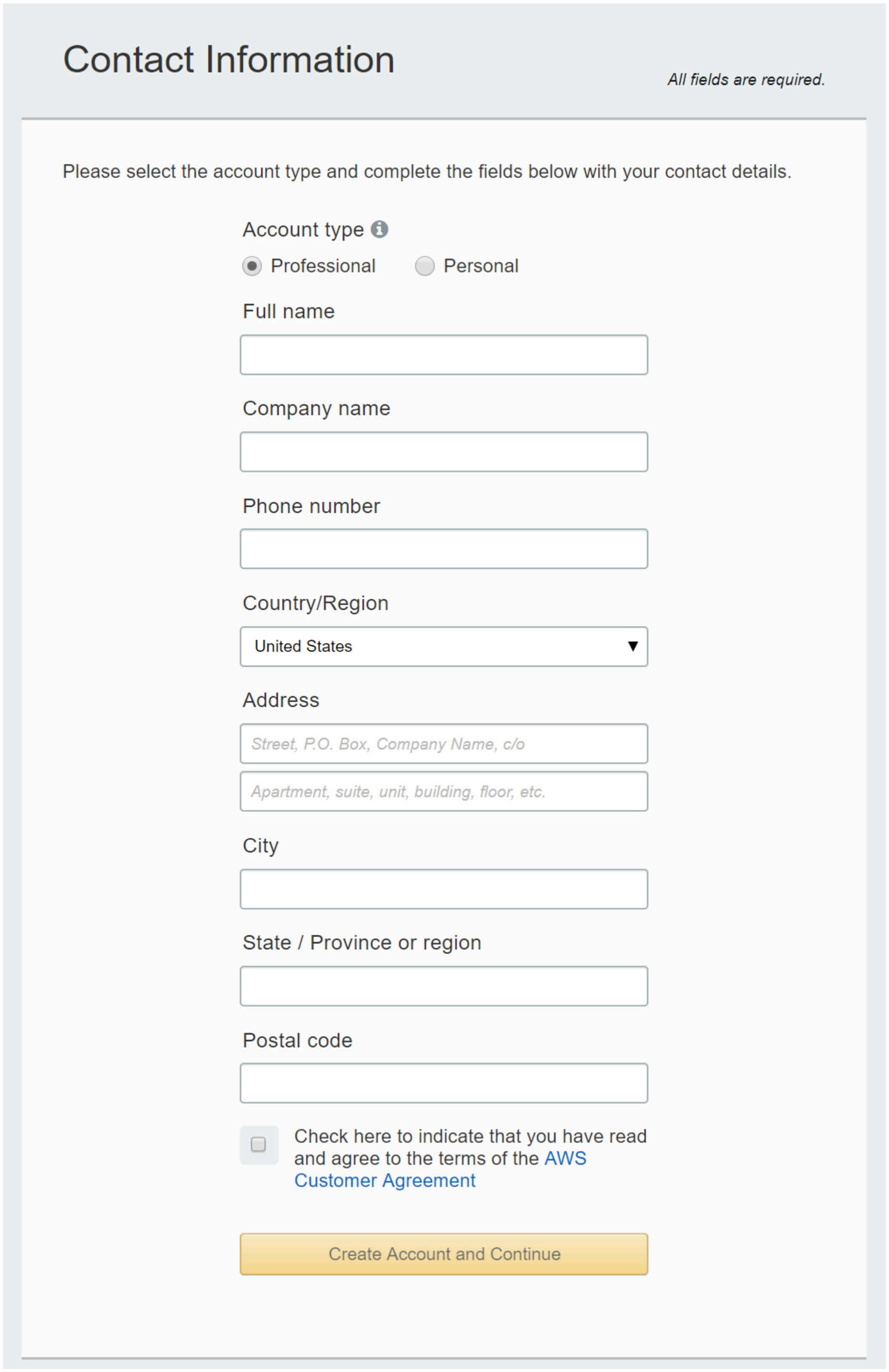
AWS Accounts Include
12 Months of Free Tier Access

Including use of Amazon EC2, Amazon S3, and Amazon DynamoDB
Visit **aws.amazon.com/free** for full offer terms

© 2018 Amazon Web Services, Inc. or its affiliates.
All rights reserved.

Initial AWS signup account details

After you enter your basic info, you'll be asked if it's a personal or professional account, with a slightly different setup for each. For professional accounts, you'll be asked for more corporate information like your company name.



Contact Information

All fields are required.

Please select the account type and complete the fields below with your contact details.

Account type

[● Professional] ○ Personal

Full name

Company name

Phone number

Country/Region

United States ▼

Address

Street, P.O. Box, Company Name, c/o

Apartment, suite, unit, building, floor, etc.

City

State / Province or region

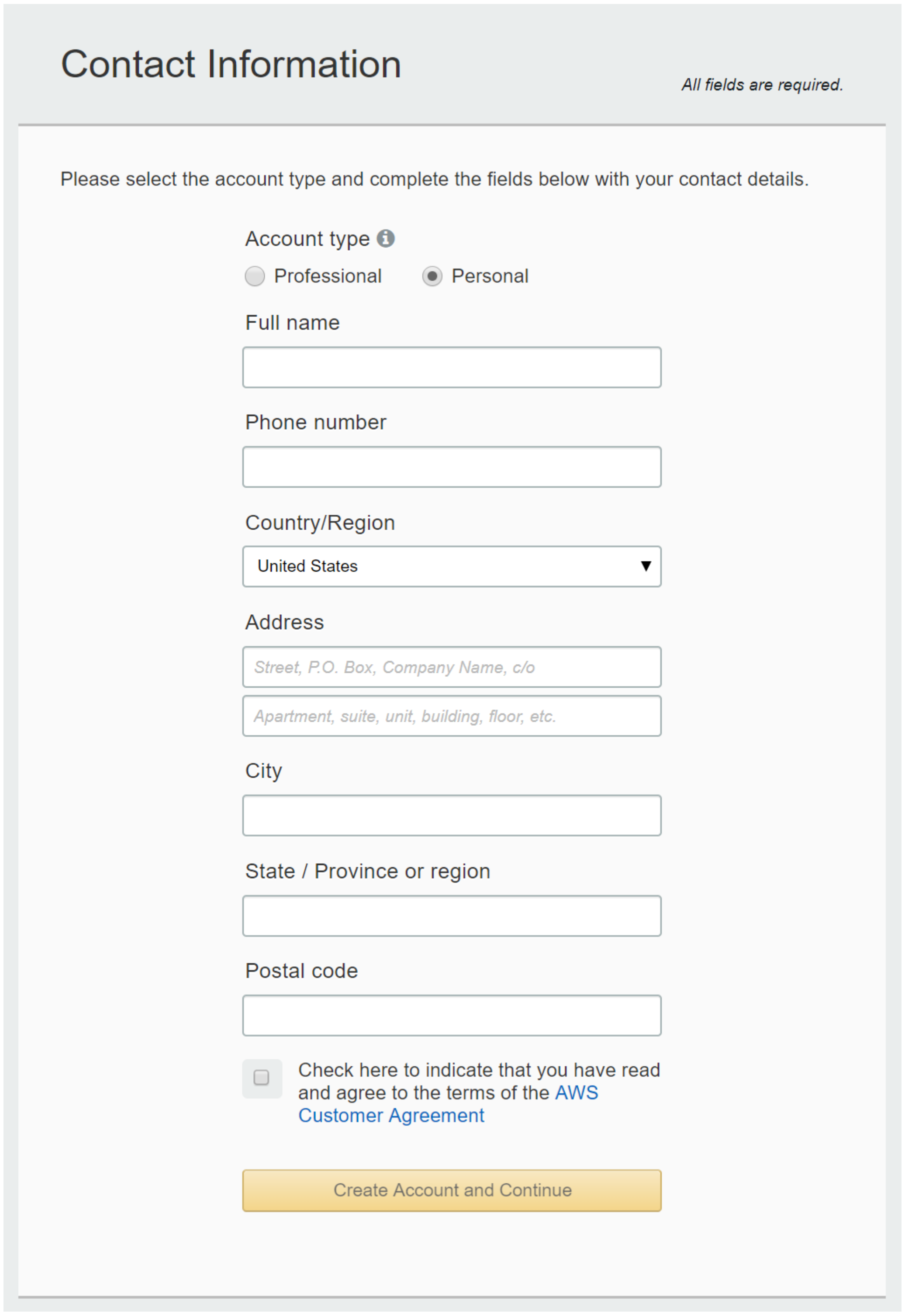
Postal code

☐ Check here to indicate that you have read and agree to the terms of the AWS Customer Agreement

Create Account and Continue

AWS detailed info professional

For personal accounts, you'll simply be asked for information strictly about yourself.



Contact Information

All fields are required.

Please select the account type and complete the fields below with your contact details.

Account type

Professional [✓ Personal]

Full name

Phone number

Country/Region: United States

Address

Street, P.O. Box, Company Name, c/o

Apartment, suite, unit, building, floor, etc.

City

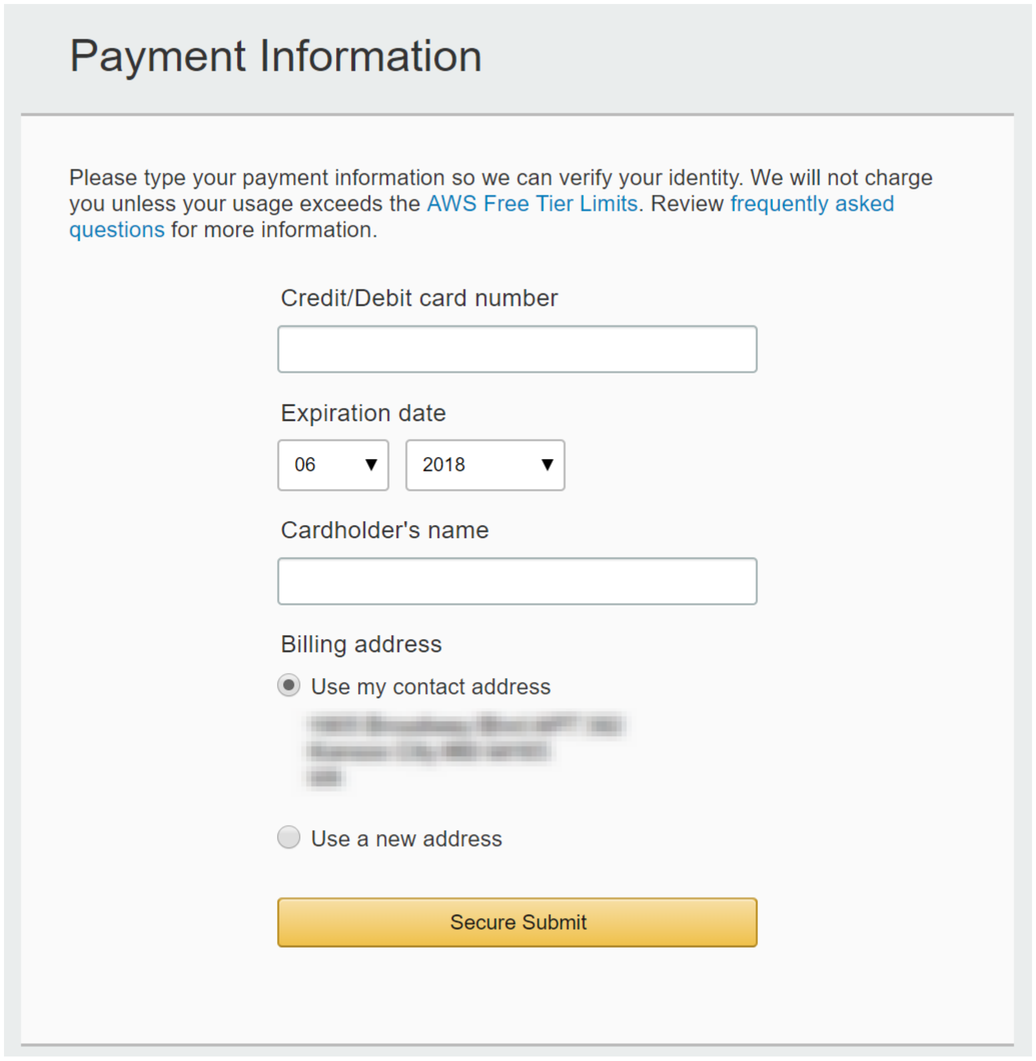
State / Province or region

Postal code

☐ Check here to indicate that you have read and agree to the terms of the AWS Customer Agreement

AWS detailed info personal

Next, you'll be asked for payment information. You'll have to provide this even for free accounts, since not all AWS services qualify for the free tier. We'll talk about the billing later, but there are a significant number of AWS resources that qualify for the free tier. Whether or not you utilize the free tier resources, you'll need to enter credit card information to proceed.



Payment Information

Please type your payment information so we can verify your identity. We will not charge you unless your usage exceeds the AWS Free Tier Limits. Review frequently asked questions for more information.

Credit/Debit card number

Expiration date: 06 2018

Cardholder's name

Billing address

[✓ Use my contact address]

Use a new address

Card info entry

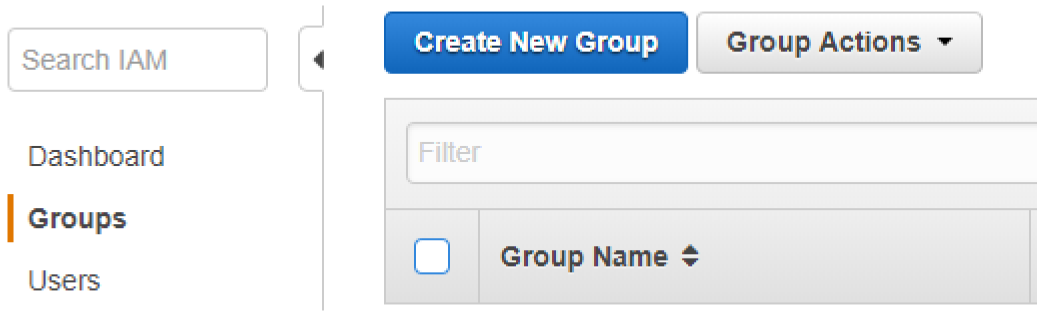
Once your card information is entered, you'll need to add your phone number, which can be verified by text code or via a phone call confirmation. Lastly, you'll be asked about AWS Support plans, where you could have up to 24/7 access to an account representative to ask all of your questions to. Once selected, your account will be provisioned and you'll be sent an email once it is ready.

Create an IAM User and Generate Access Keys

The next thing we'll want to do is create a user within the Identity and Access Management (IAM) service and then generate some access keys so you can manage and operate your AWS resources from PowerShell. Hopefully this will be the last thing that we *have* to do via the AWS Management Console. I'll try and keep it strictly PowerShell focused from here on out.

For your first user, what I'd recommend doing is create an "Administrators" group with a user named "Administrator". This prevents you from making any changes with your root account credentials (since it's been a best practice for a good long while to not use root for everything) as well as it gives you a superuser account that's easy to shut down in case your credentials to the account get leaked.

The first thing we'll need to do to get the account set up is to log into the AWS Management Console and navigate to the "IAM" service. Once there, select "Groups" then "Create new Group"

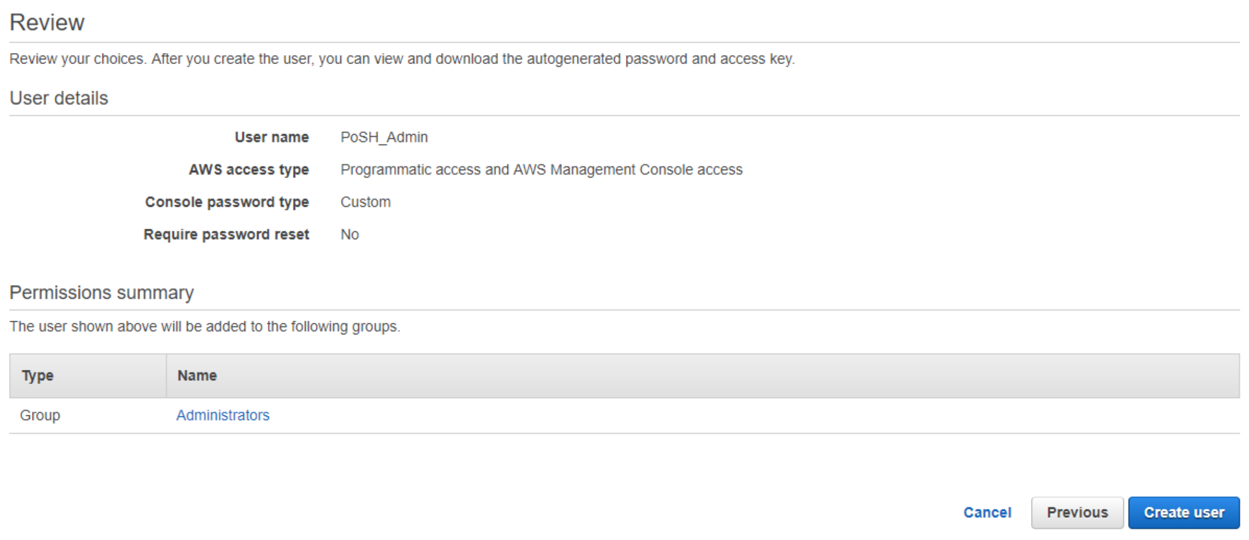


Select "Create new Groups" in the IAM service

Once selected, you'll name your Administrators group however you see fit. Mine is simply named "Administrators" but depending on your organizations standards you may have aspecial nomenclature you need to follow for it. After naming your group, you'll need to select some permissions for it. In this case, since we're making a group for our Administrators, we'll select the pre-built "AdministratorAccess" default policy.

Now we need to add a user to the group, so we'll click on the "Users" tab on the left side, and click "Create User". We'll then need to name our user and select whether the user can log in to the AWS Management Console or have programmatic access, or both. If you're new to AWS, it may be beneficial at this point for you to select both, since you'll be able to log in and verify the changes you're making with PowerShell. By selecting AWS Management Console access your account will have a password generated for it. If you feel that you're a seasoned veteran and have a handle on what's going on in AWS (or just really, *really*, love your `Get-*` commands) you can select programmatic access only. This'll generate the key pair we need to authenticate to AWS with PowerShell.

Once we've selected the initial settings, we'll click "next" and add the user to our "Administrators" group. Before creating the user, take one quick second to verify that the users details are correct.



Review

Review your choices. After you create the user, you can view and download the autogenerated password and access key.

User details

User name	PoSH_Admin
AWS access type	Programmatic access and AWS Management Console access
Console password type	Custom
Require password reset	No

Permissions summary

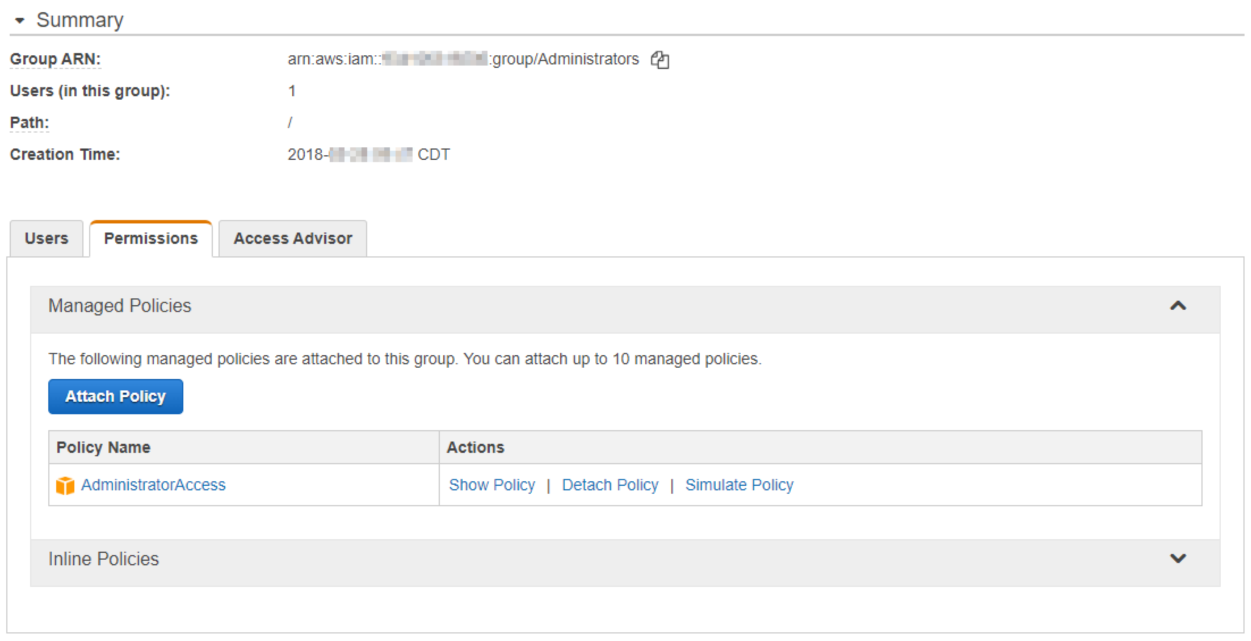
The user shown above will be added to the following groups.

Type	Name
Group	Administrators

User Setting verification

Once your user is created, you'll be taken to a screen where you can download a CSV with the users credentials. Make sure to do this, because if you're at all like me you'll just plow through the menu and forget to grab the secret access key. The secret access key is the one that can't be generated again, so you'd have to make a new key pair for the user if you lose it (just don't lose it, it's a pain).

If you want to verify everything, you can navigate back to the groups tab and your group should look something like this:



Summary

Group ARN:	arn:aws:iam::[redacted]:group/Administrators
Users (in this group):	1
Path:	/
Creation Time:	2018-[redacted] CDT

Managed Policies

The following managed policies are attached to this group. You can attach up to 10 managed policies.

Policy Name	Actions
AdministratorAccess	Show Policy | Detach Policy | Simulate Policy

Inline Policies

Administrators Group Settings

Installing AWS PowerShell for Windows

Now that you have your account created and the credentials saved, let's install the AWS PowerShell module. This section is on how to install the module on Windows, but I've included a section on Linux and MacOS below.

For starters, we need to determine which version of the AWS Tools for Windows PowerShell to install. This is largely determined by the PowerShell version we're running on our computer. You can find the major version of PowerShell by running the below command:

```
PS> $PSVersionTable.PSVersion.Major
```

This will return a number between 2-6, and you can follow the instructions for installing the AWS Tools for Windows PowerShell based on which version you're running. As of Windows 10, the default version of PowerShell is 5 or 5.1, so for the vast majority of cases you'll be able to use the instructions below without needing to download and install the Windows Management Framework, but I wanted to provide those instructions just in case.

For PowerShell Version 2-5

To start with the installation of the AWS PowerShell module for older versions of PowerShell, we'll need to determine if we need to install the Windows Management Framework first. If you're using Windows XP or Vista, you'll need to first install the Windows Management Framework before you can proceed. Follow this Link[4] to download the WMF files and follow the installation instructions.

Installing AWS Tools for Windows PowerShell with the MSI

Once you have the Windows Management Framework installed, or if you're using Windows 7 or later, you can simply install the AWS Tools for Windows PowerShell via the provided MSI. Go Here[5] to download the MSI for your system's architecture. Once downloaded, you can simply double click the MSI, or use the following command, changing the path to your downloaded msi appropriately:

```
PS> msiexec.exe /qn /i AWSToolsAndSDKForDotNet.msi
```

Installing AWS Tools for Windows PowerShell with the PowerShell Gallery

Another convenient way to install the AWS PowerShell module is to install it via the PowerShell Gallery. In order to install this way, you can simply run the following command from an elevated command prompt (additionally, you can use the scope "CurrentUser" to not require administrative privileges).

[4]http://support.microsoft.com/kb/968929
[5]https://aws.amazon.com/powershell/

```
PS> Install-Module -Name AWSPowerShell -Scope AllUsers
```

Depending on the trust relationships you or your organization has set up, you may get the following message:

```
Untrusted repository
You are installing the modules from an untrusted repository. If you trust this repos\
itory, change its
InstallationPolicy value by running the Set-PSRepository cmdlet. Are you sure you wa\
nt to install the modules from
'PSGallery'?
[Y] Yes  [A] Yes to All  [N] No  [L] No to All  [S] Suspend  [?] Help (default is "N\
"):
```

Go ahead and press "A" to install the module.

For PowerShell Version 6

The easiest way to install the AWS PowerShell module for PowerShell version 6 is to use the PowerShell Gallery. We do have to specify a slightly different module than in older versions of PowerShell, since this one relates specifically to the .NET Core version of AWS PowerShell. Similarly, though, you'll need to run this in an elevated PowerShell session since we're installing it for all users (you can use the scope "CurrentUser" to not require administrative privileges).

```
PS> Install-Module -Name AWSPowerShell.NetCore -AllowClobber -Scope AllUsers
```

Much like in the previous versions, if you haven't trusted the repository yet you'll get the following message. Select "A" to continue to install the module

```
Untrusted repository
You are installing the modules from an untrusted repository. If you trust this repos\
itory, change its
InstallationPolicy value by running the Set-PSRepository cmdlet. Are you sure you wa\
nt to install the modules from
'PSGallery'?
[Y] Yes  [A] Yes to All  [N] No  [L] No to All  [S] Suspend  [?] Help (default is "N\
"):
```

Installing AWS PowerShell for Linux/MacOS

Since PowerShell is now a truly cross platform scripting language, installing the AWS PowerShell module on Linux or MacOS is a reasonable thing for an administrator to do. There is one caveat that Linux and MacOS users can only use the AWS PowerShell Tools for .NET Core, but this is hardly a limitation. In order to get started installing the AWS PowerShell module on your Linux or Mac machine, we'll need to follow the steps below:

- Install PowerShell on your device (since there are many different flavors, I'll let you refer to your OS documentation on how to install PowerShell)
- Start PowerShell by running pwsh in your system's terminal

Once we have PowerShell open, our installation instructions may look pretty familiar. We'll run the below PowerShell command to install the module:

```
$ pwsh
PS> Install-Module -Name AWSPowerShell.NetCore -AllowClobber
```

This will install the module for the local user, but if you're wanting to install it for all users on the machine, you'll need to relaunch your `pwsh` terminal with `sudo` privileges and add the `-Scope AllUsers` parameter.

Importing the Module

Installing the module isn't quite enough with PowerShell. Now that it's installed, it's simply available to us. We still need to import the module in our script or in our PowerShell session in order to use the cmdlets available. By using the `Import-Module` cmdlet, we are able to load the module into memory in order to call the available cmdlets. I've written this one liner to import the version of AWSPowerShell installed on your machine, but you can also specify the Module you want to import by using the `Import-Module` cmdlet alone.

```
PS> Get-Module AWSPowerShell* | Import-Module
```

Now that will take care of you for this session, but what if you use PowerShell every day, and everything you write is going to be about AWS? Well in that case, you may want to go ahead and update your PowerShell Profile.

To start with this, we'll first need to determine where your PowerShell profile is located. You can simply call the `$profile` variable, and PowerShell will tell you where to go. If you like to use Visual Studio Code as your editor, within PowerShell you can type the following:

```
PS> code $profile
```

This will open up your PowerShell Profile in the Visual Studio Code editor. Alternatively, if you don't use VSCode, you may need to navigate to the folder, and open up the `profile.ps1` file that exists there.

Once opened, you'll add the following line to your profile:

```
Get-Module AWSPowerShell* | Import-Module
```

Now, every time you go to open a new PowerShell window, you'll first import the module before you're even greeted with the prompt. **PLEASE NOTE**: This adds a little bit of time for your window to come up and let you start typing. If performance is of the essence, you may just want to manually import the module as you need it rather than loading it automatically every time.

Configuring AWS PowerShell

Now that the PowerShell module is installed and imported, and we have our access keys created, let's configure the module to be able to work with our account.

Setting up a the Default Profile

Remember that handy CSV of the account credentials we made from earlier? Now's the time to bring it back out. We're going to configure our default AWS Profile with these credentials. AWS Profiles are what we use to authenticate as various users or roles within our AWS environment. For this case, we're going to set up our Administrator's credentials, but you will need to remember these or come back and reference this documentation since you'll be using this a lot in your AWS journey.

To configure the default profile, we'll use the following commands, replacing the Access Key and Secret Key with the respective key in your CSV you saved:

```
PS> $AccessKey = "YOUR_ACCESS_KEY_HERE"
PS> $SecretKey = "YOUR_SECRET_KEY_HERE"
PS> Set-AWSCredential -AccessKey $AccessKey -Secretkey $SecretKey
```

You can verify that this was successful by running the next cmdlet:

```
PS> Get-AWSCredential -ListProfileDetail

ProfileName StoreTypeName         ProfileLocation
----------- -------------         ---------------
default     NetSDKCredentialsFile
```

And alternatively, if you don't want to keep your credentials, or if you goofed something up and the credentials don't work, you can always use the following to get rid of them from your machine by running the following:

```
PS> Remove-AWSCredentialProfile -ProfileName default
```

For information about using multiple profiles, check out Chapter 3: Authentication

Specifying Region

Another useful feature when setting up our credentials is to also set another preference: our preferred region. In this case, we're going to use my personal favorite of "us-west-2" which is the Oregon region. Don't worry, you can always change this and you can always specify the `-Region` parameter in all of your scripts if you want to make sure to use a particular region. Specifying a default region prevents that parameter from being necessary by default.

```
PS> Set-DefaultAWSRegion -Region "us-west-2"
```

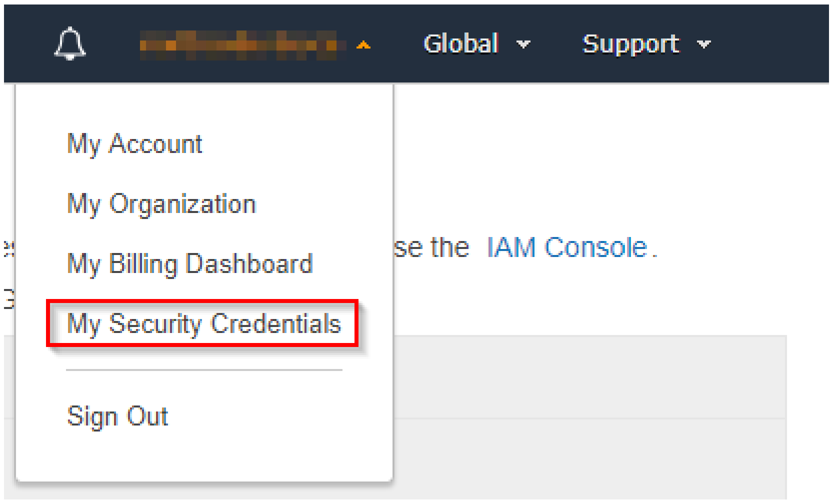
Of course, there's an accompanying `Get-DefaultAWSRegion` cmdlet, so if you ever forget what you set yours as, or if you want to verify that the change took place, you can always use that cmdlet.

AWS Account Best Practices

We're really cruising through this setup. I hope you're getting excited to see the really fun stuff that we can do with AWS PowerShell. We've set up our AWS account, created our administrative user, installed the AWS PowerShell module, imported it, set up our credentials from the user we created, and now we are almost ready to go. Before we get to the really fun stuff, let's take a moment to make sure that your account is safe and secure, with appropriate logging and limits set in place.

Enable 2 Factor Authentication

The first thing that we should do to take steps toward securing our account is to enable multi-factor authentication (MFA) for your root account. One thing to note, is that you can only configure one multi-factor authentication device per root or IAM user account, so make sure as you set it up that it's a device that you'll continuously have access to. In order to enable this, first you'll need to log into your account, then we'll need to navigate to the "Security Credentials" page, by clicking on your username in the top right corner of the AWS Management Console, then selecting "My Security Credentials".



Select "My Security Credentials"

Once selected, you'll click to activate your MFA system of choice, and follow the steps to complete the setup for your selection.

Set Password Policy

The next thing to do in order to secure your account is to set a password policy for your IAM users. Although this won't change much for passwords that are already set, this will set a standard for going forward. With the password policy, we can set the following options:

- Minimum password length
- Required character types, like uppercase and lowercase letters, symbols, and numbers.
- Whether or not IAM users are allowed to reset their own passwords
- Password expiration time periods
- Previous password allowance
- and more

Depending on your organizations security standards, you should set the password policy accordingly. In order to set it, let's pop open the closest PowerShell window to view and then set the policy. Viewing the policy is pretty easy. You'll find that the AWS PowerShell team has made all of the cmdlets according to PowerShell best practice, so you have your standard `Get-*`, `Set-*`, and `Update-*` style of commands. To retrieve the current password policy, we'll run the following:

```
PS> Get-IAMAccountPasswordPolicy

AllowUsersToChangePassword : True
ExpirePasswords            : False
HardExpiry                 : False
MaxPasswordAge             : 0
MinimumPasswordLength      : 8
PasswordReusePrevention    : 0
RequireLowercaseCharacters : True
RequireNumbers             : True
RequireSymbols             : True
RequireUppercaseCharacters : True
```

If you haven't changed anything yet, you should have a response that's pretty similar to mine. In order to change this, we'll use the cmdlet below, specifying the parameters for the items you'd like to change in your password policy. By using the backtick at the end of a line, you can escape the carriage return, making your commands significantly more human readable. You could also use other PowerShell methods like splatting in order to fill in the cmdlet's parameter values.

```
PS> Update-IAMAccountPasswordPolicy `
>       -AllowUsersToChangePassword $true `
>       -HardExpiry $false ` # $true prevents users from resetting passwords after i\
t has expired on their own
>       -MaxPasswordAge 90 ` # days
>       -MinimumPasswordLength 10 ` #characters
>       -PasswordReusePrevention 3 `
>       -RequireLowercaseCharacter $true `
>       -RequireNumber $true `
>       -RequireSymbol $true `
>       -RequireUppercaseCharacter $true
```

Once we've updated the policy, let's run the `Get` command to see the new password policy in action.

```
PS> Get-IAMAccountPasswordPolicy

AllowUsersToChangePassword : True
ExpirePasswords            : True
HardExpiry                 : False
MaxPasswordAge             : 90
MinimumPasswordLength      : 10
PasswordReusePrevention    : 3
RequireLowercaseCharacters : True
```

```
RequireNumbers             : True
RequireSymbols             : True
RequireUppercaseCharacters : True
```

That's all there is to updating the policy! This applies for all IAM users across your AWS environment.

Set AWS Budget

Keep in mind that best practices aren't always about security. Another item we need to think about is our budget! A best practice is to set the maximum allowable spend in AWS. Personally, I think it is easier to set this up via the AWS Management Console, but you didn't buy this book to learn about that. Let's use PowerShell to set our AWS Budget. There are a significant number of parameters we need to worry about here, so I'm going to use PowerShell's splatting feature to make sure they all go in the right place. I won't go in to each parameter's details, but I will give you a highlight of a few of the major ones.

- `-AccountId` - This is the support account ID that is generated by AWS. To find this, you'll use the AWS Management Console, click "Support" in the upper right hand corner, then "Support Center". Your account ID will be clearly labeled at the top.
- `-BudgetLimit_Amount` - This is the amount you're willing to spend on AWS in the timeframe that you'll specify in just a minute. If you want to spend $200 US Dollars in a month, this value would be '200' for you.
- `-Budget_BudgetType` - There are a couple of different types of budgets that you can create with these cmdlets. You can create resource utilization budgets, you can create resource usage budgets, or you can create cost budgets. For our purposes, since we're wanting to limit our spend, we will select 'COST'
- `-Budget_TimeUnit` - This is how often you "reset the clock" on your budget. Maybe you know you only want to spend $2,000 a year, so you could put "ANNUALLY". For me, I like to track my budgets monthly so that's the timeframe I'll go with. Quarterly and Daily are also options, if that works better for your business model.
- `-BudgetLimit_Unit` - Finally we need to specify what currency we're actually spending (and it turns out 'Monopoly' is an invalid unit for this parameter - go figure). Since I'm from the US, I prefer all of my spending to be done in USD, but you can use other currency denominations as well.

For an example of how to set up a budget, I've set a little $10 monthly max spend cap on my free tier account, just so I don't go crazy writing this book. You can use this as a template for your personal spend below:

```
$BGTParams = @{
    'AccountId' = '934106316506';
    'BudgetLimit_Amount' = '10';
    'Budget_BudgetName' = 'CorporateLimit';
    'Budget_BudgetType' = 'COST';
    'Budget_TimeUnit' = 'MONTHLY';
    'BudgetLimit_Unit' = 'USD';
}

New-BGTBudget @BGTParams
```

Summary

I hope you've had fun getting your AWS account and development environment all set up. You've officially gotten your feet wet with this amazing system of tools that have been provided by Amazon, and you've laid a great foundation for continuing on in your education about AWS PowerShell. This is about as basic as it gets for this book, and we'll dive deeper into the AWS cmdlets, services, and PowerShell features as we continue on.

Next, we'll start looking at the fundimentals of the AWS PowerShell module's structure, how to look up cmdlets, and even how to convert AWS CLI scripts into PowerShell.

Chapter 2: Getting Started using PowerShell on AWS

Now that we've gotten set up with our account, created our access keys, and ran through some of the best practices, we can get into the juicy bits of AWS and PowerShell. This will be the traditional "first chapter" where we learn how to learn more about the PowerShell cmdlets and options that are available to us on AWS. I always refer back to "the bible" of PowerShell, *Learn PowerShell In a Month of Lunches* by Don Jones and Jeff Hicks, where they talk about the `Get-Help` and `Get-Command` cmdlets and how beneficial they are as a building block to learning the rest of PowerShell. Similarly, in this book we'll go over the traditional cmdlet discovery methods, and then show you some of the new tricks that are built into AWS. We'll then go into detail by learning about the pipeline and how it relates to AWS PowerShell commands.

Traditional Cmdlet Discovery

The first think we should do, now that we have the AWS PowerShell installed and configured on our machine, is to start looking for the cmdlets available to us. Naturally, we have the "normal" PowerShell way of doing this.

```
PS> Get-Command -Module AWSPowerShell | Select-Object -first 20
```

If you're following along on your own machine, I *strongly* recommend using some kind of filtering such as the `Select-Object` filter to grab the first 20 results of this, because there are simply *so many cmdlets* that AWS provides in their PowerShell module. This is a great thing in reality, because it allows us to manage every aspect of our AWS environment with PowerShell, but it also makes these function which return everything take quite some time. AWS also has a standard nomenclature for their cmdlets, so if you have a specific service that you want to see the cmdlets for, you can surround the service abbreviation in wildcards in order to find all of the cmdlets specific to that service. For example, if you want to see all of the cmdlets relating to the "EC2" service, you could use `Get-Command` like so.

```
PS> Get-Command -Module AWSPowerShell *EC2*
```

It will return the following results:

```
CommandType     Name                                               Version    Source
-----------     ----                                               -------    ------
Alias           Edit-EC2Hosts                                      3.3.225.1  AWSPow\
erShell
Alias           Get-EC2AccountAttributes                           3.3.225.1  AWSPow\
erShell
Alias           Get-EC2ExportTasks                                 3.3.225.1  AWSPow\
erShell
Alias           Get-EC2FlowLogs                                    3.3.225.1  AWSPow\
erShell
Alias           Get-EC2Hosts                                       3.3.225.1  AWSPow\
erShell
Alias           Get-EC2ReservedInstancesModifications              3.3.225.1  AWSPow\
erShell
Alias           Get-EC2VpcPeeringConnections                       3.3.225.1  AWSPow\
erShell
Alias           New-EC2FlowLogs                                    3.3.225.1  AWSPow\
erShell
Alias           New-EC2Hosts                                       3.3.225.1  AWSPow\
erShell
Alias           Remove-EC2FlowLogs                                 3.3.225.1  AWSPow\
erShell
Cmdlet          Add-EC2ClassicLinkVpc                              3.3.225.1  AWSPow\
erShell
Cmdlet          Add-EC2InternetGateway                             3.3.225.1  AWSPow\
erShell
Cmdlet          Add-EC2NetworkInterface                            3.3.225.1  AWSPow\
erShell
Cmdlet          Add-EC2Volume                                      3.3.225.1  AWSPow\
erShell
...
```

If you try this out on a few different services, you'll find that the native PowerShell command isn't perfect. Looking at a service such as the Systems Manager (abbreviated "SSM" in the AWS PowerShell module) you'll end up with a bunch of results that aren't quite what you're looking for.

```
PS> Get-Command -Module AWSPowerShell -Name *SSM*

CommandType     Name                                           Version    Source
-----------     ----                                           -------    ------
Alias           Get-SNSSMSAttributes                           3.3.225.1  AWSPow\
erShell
Alias           Get-SSMMaintenanceWindowTargets                3.3.225.1  AWSPow\
erShell
Alias           Get-SSMParameterNameList                       3.3.225.1  AWSPow\
erShell
Alias           Set-SNSSMSAttributes                           3.3.225.1  AWSPow\
erShell
Cmdlet          Add-SSMResourceTag                             3.3.225.1  AWSPow\
erShell
Cmdlet          Edit-SSMDocumentPermission                     3.3.225.1  AWSPow\
erShell
Cmdlet          Get-DMSReplicationTaskAssessmentResult         3.3.225.1  AWSPow\
erShell
Cmdlet          Get-INSAssessmentReport                        3.3.225.1  AWSPow\
erShell
Cmdlet          Get-INSAssessmentRun                           3.3.225.1  AWSPow\
erShell
Cmdlet          Get-INSAssessmentRunAgent                      3.3.225.1  AWSPow\
erShell
Cmdlet          Get-INSAssessmentRunList                       3.3.225.1  AWSPow\
erShell
Cmdlet          Get-INSAssessmentTarget                        3.3.225.1  AWSPow\
erShell
Cmdlet          Get-INSAssessmentTargetList                    3.3.225.1  AWSPow\
erShell
Cmdlet          Get-INSAssessmentTemplate                      3.3.225.1  AWSPow\
erShell
...
```

It looks like we have quite a few odd ducks in there. As you can see, the filter we've applied narrows it down to anything with "SSM" sequentially in the cmdlet name. We have a few cmdlets being returned for the Simple Notification Service (SNS), we have a few returned for the AWS Inspector (INS) and we even have one for the Database Migration Service (DMS). The sheer number of cmdlets cause for some collision when you use wildcards with the traditional cmdlet discovery functions. You can fix this a little more by adding the hypen before SSM, because according to the AWS nomenclature, the "Verb" in the PowerShell cmdlet always starts with the service abbreviation, but having to remember this syntax before you query for cmdlets is cumbersome, especially when you then have to sift through the long list of cmdlets returned to find the function you're actually looking for.

Built-In Discovery Cmdlet

One of the ways built into the AWS PowerShell module to discover cmdlets is to use the function `Get-AWSCmdletName`. This little genius has three ways that are incredibly useful to quickly find and discover AWS PowerShell cmdlets relating to AWS resources.

Query by Service Name

First, we can use the `-ServiceName` parameter to find cmdlets relating to a specific service. If we take our “SSM” example from the previous section and play it here, we can find that we can get very accurate results back quickly.

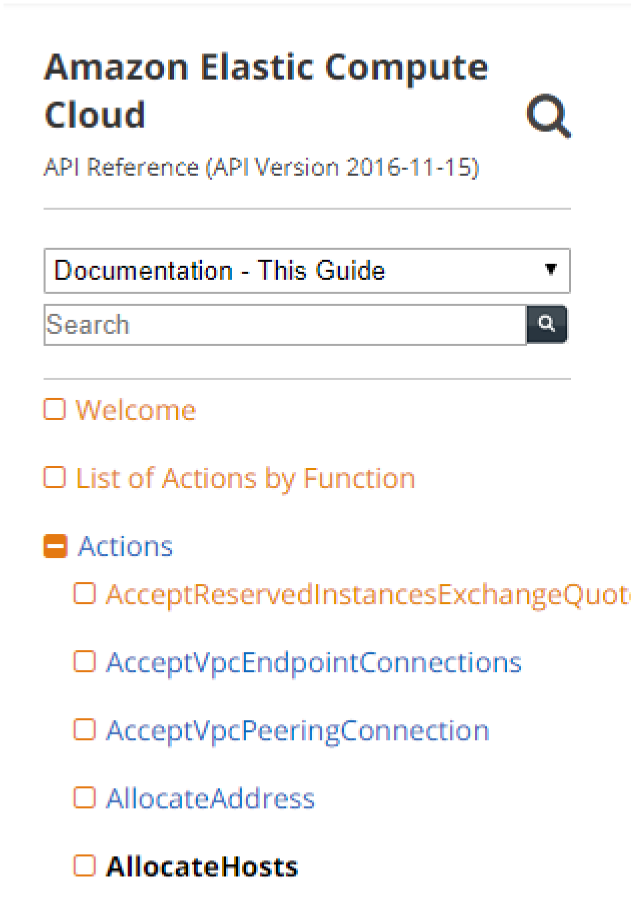
```
PS> Get-AWSCmdletName -Service SSM

CmdletName                        ServiceOperation                    ServiceName
----------                        ----------------                    -----------
Add-SSMResourceTag                AddTagsToResource                   Amazon Simple \
Systems Management
Edit-SSMDocumentPermission        ModifyDocumentPermission            Amazon Simple \
Systems Management
Get-SSMActivation                 DescribeActivations                 Amazon Simple \
Systems Management
Get-SSMAssociation                DescribeAssociation                 Amazon Simple \
Systems Management
Get-SSMAssociationList            ListAssociations                    Amazon Simple \
Systems Management
Get-SSMAssociationVersionList     ListAssociationVersions             Amazon Simple \
Systems Management
Get-SSMAutomationExecution        GetAutomationExecution              Amazon Simple \
Systems Management
Get-SSMAutomationExecutionList    DescribeAutomationExecutions        Amazon Simple \
Systems Management
Get-SSMAutomationStepExecution    DescribeAutomationStepExecutions    Amazon Simple \
Systems Management
...
```

As you can see, this is only returning results related to the Simple Systems Management (SSM) service. Since all of the commands that we use for AWS relate to a specific AWS service, this becomes a very useful tool in narrowing down the number of commands that we need. From here, we can of course use all of our familiar PowerShell-y commands which let us sort, filter, and narrow down our results.

Query by API Operation

My next favorite parameter to use with this cmdlet is `-ApiOperation`. This will relate our query to the .NET API that the module is built on top of, and filter back results based on the API operation it would perform. Looking through AWS's documentation, and looking through the documentation of your developers, you'll get a sense for how prevalent references to the API become. Keeping in mind that the AWS PowerShell module at its core is calling the API through the .NET SDK, you will be able to query the PowerShell module to see if a cmdlet exists to call on the operations you come across in documentation.

Let's have an example! I navigated to the AWS EC2 API Reference Pages[6] and started poking around. Since the Elastic Compute Cloud (EC2) service is pretty easy to wrap our heads around (think traditional VM's hosted in the cloud) let's dig a little deeper to see what all we can do with it.


Amazon Elastic Compute Cloud
API Reference (API Version 2016-11-15)
Documentation - This Guide
Search
Welcome
List of Actions by Function
Actions
AcceptReservedInstancesExchangeQuot
AcceptVpcEndpointConnections
AcceptVpcPeeringConnection
AllocateAddress
AllocateHosts

Navigating through the AWS EC2 API Reference

Once here, open up the "Actions" menu, and select "AllocateHosts". This will give us more details on the AWS API operation to allocate new EC2 instances. Since we know that we can do this with the API, we ought to also be able to do this with the PowerShell module, right? Using PowerShell,

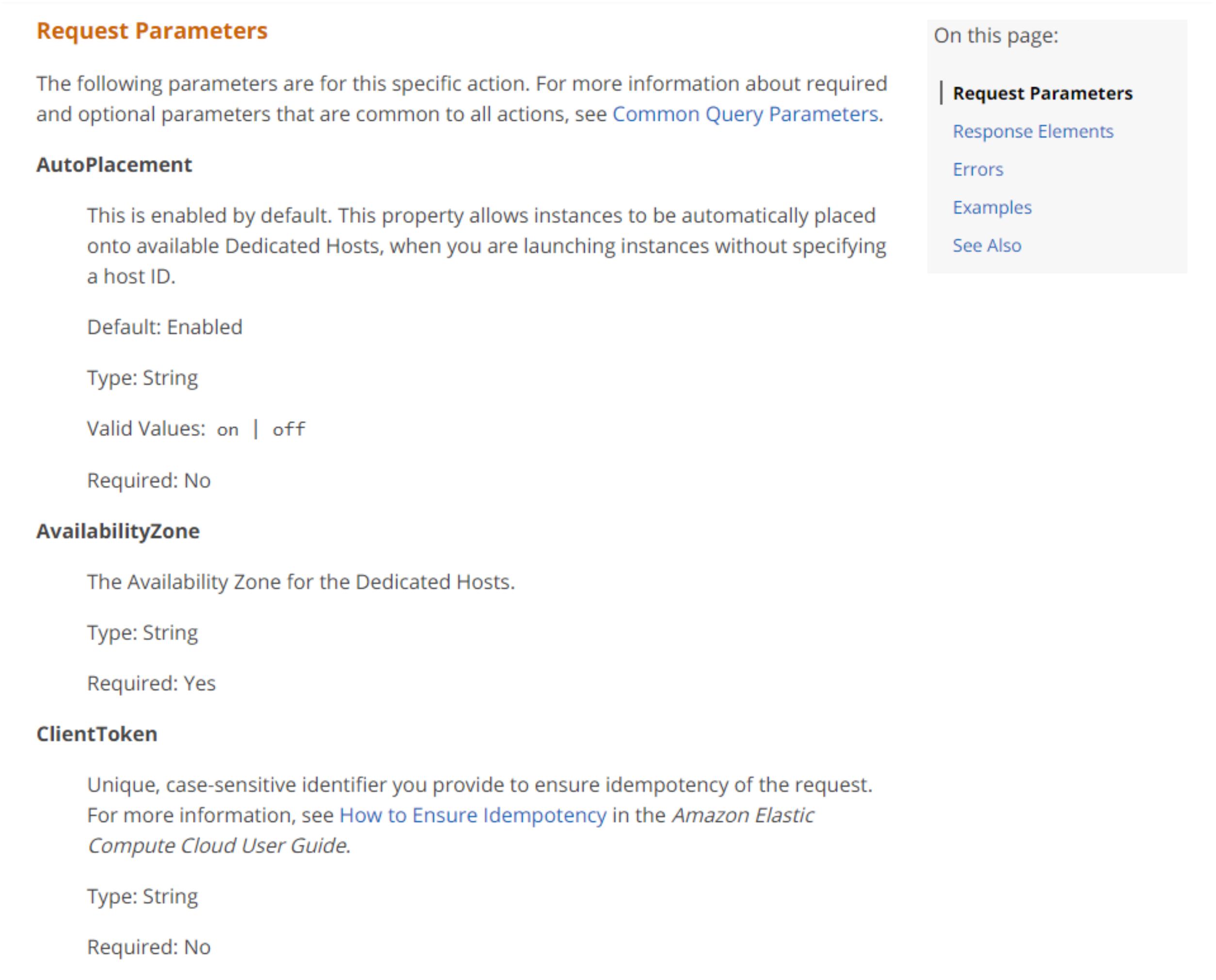
[6]https://docs.aws.amazon.com/AWSEC2/latest/APIReference/Welcome.html

we can use the `-ApiOperation` parameter to get a succinct result back, as seen below.

```
PS> Get-AWSCmdletName -Service EC2 -ApiOperation AllocateHosts

CmdletName  ServiceOperation ServiceName                  CmdletNounPrefix
----------  ---------------- -----------                  ----------------
New-EC2Host AllocateHosts    Amazon Elastic Compute Cloud EC2
```

Just like that, we can see that by using the `New-EC2Host` cmdlet, we'd be able to call the AllocateHosts API Operation in order to provision a new EC2 instance. What's really fun is to read through the online API documentation's "Request Parameters" section to read about the individual parameters, and then compare it with what we get back from the command below (Spoiler alert, they're the same):

```
PS> Get-Help New-EC2Host

NAME
    New-EC2Host

SYNOPSIS
    Calls the Amazon Elastic Compute Cloud AllocateHosts API operation.


SYNTAX
    New-EC2Host [-AutoPlacement <Amazon.EC2.AutoPlacement>] [-AvailabilityZone <Syst\
em.String>] [-ClientToken <System.String>] [-InstanceType <System.String>] [-Quantit\
y
    <System.Int32>] [-Force <System.Management.Automation.SwitchParameter>] [<Common\
Parameters>]
...
```



Request Parameters

The following parameters are for this specific action. For more information about required and optional parameters that are common to all actions, see Common Query Parameters.

AutoPlacement

This is enabled by default. This property allows instances to be automatically placed onto available Dedicated Hosts, when you are launching instances without specifying a host ID.

Default: Enabled

Type: String

Valid Values: `on | off`

Required: No

AvailabilityZone

The Availability Zone for the Dedicated Hosts.

Type: String

Required: Yes

ClientToken

Unique, case-sensitive identifier you provide to ensure idempotency of the request. For more information, see How to Ensure Idempotency in the *Amazon Elastic Compute Cloud User Guide*.

Type: String

Required: No

Request Parameters online EC2 API documentation reference

Query by Command Line Interface Command

We can also get cmdlets by using my personal favorite parameter, the `-AwsCliCommand`. As you could imagine, this allows you to type in an AWS CLI command name, and you'll get the PowerShell equivalent. If your current AWS operations team uses the AWS CLI, this is a quick and easy way to get started converting their scripts to PowerShell. There's one small caveat, however, in that not *all* AWS CLI commands relate to a PowerShell command. There are simply certain things that the CLI can do that PowerShell can't. One example of this is the very powerful `aws s3 sync` command, which essentially runs the linux command `rsync` between your local machine and/or multiple s3 buckets. If you're not familiar with `rsync`, it allows you to synchronize all of the contents of one folder to another.

We can find some great examples of this query in use, however, and let's use AWS S3 as the service we want to play with. You'll end up copying a lot of data into S3 over the course of your career with AWS, so it's only reasonable to assume that you'll have a few scripts set up which will take care of some data transfers. What if we were wanting to convert these AWS CLI scripts (or even one-liners stored in your history!) into PowerShell? Let's play around with this, for instance, if we were wanting to list all of our S3 buckets, we could use the command `aws s3 list-buckets`, but what could we use with PowerShell? Running the command will show us:

```
PS> Get-AWSCmdletName -AwsCliCommand "aws s3 list-buckets"

CmdletName   ServiceOperation ServiceName                  CmdletNounPrefix
----------   ---------------- -----------                  ----------------
Get-S3Bucket ListBuckets      Amazon Simple Storage Service S3
```

And sure enough, using the `Get-S3Bucket` cmdlet will list our S3 buckets relating to our account:

```
PS> Get-Help Get-S3bucket

NAME
    Get-S3Bucket

SYNOPSIS
    Lists your Amazon S3 buckets.
```

Pipelining Commands

The last thing that is necessary to cover with any PowerShell topic is the pipeline. Pipelining commands works with AWS PowerShell the way that it does with all other modules. There are a few pitfalls that are easy to fall into, especially when it comes to object types, but as long as you're looking out for them and are aware that they exist you should be able to make it through the pipe just fine.

One of the key things to understand as we use the pipe with AWS and PowerShell is that matching object types can be really difficult and frustrating at times. When Amazon was creating the .NET API for AWS, they created a significant number of custom object types that don't fully take advantage of the built-in object types of PowerShell.

One such instance comes to mind with dates. In PowerShell, we're used to the `TypeName: System.DateTime` object type, however, when using the AWS PowerShell command `Get-CECostandUsage`. This cmdlet, instead of using a start and end `System.DateTime` object, uses a hashtable that is it's own object type of `TypeName: Amazon.CostExplorer.Model.DateInterval`. The key to avoiding frustration when dealing with this is to refer to the documentation, pipe your output to `Get-Member` when using the IDE, and go slow and steady.

Chapter 3: Authentication

So I'll admit this is probably coming a little later than I'd like for it to in this book, but authentication is a crucial piece of managing your AWS infrastructure, whether you're operating the environment alone or with a group. Being an infosec guy at heart, I am constantly thinking about the "CIA Triangle" of Confidentiality, Integrity, and Accessibility. Those three sides of the triangle work together to form who can access what resources, when, and how. By properly utilizing the IAM module of AWS, and with the appropriate level of security and controls around your access keys, you can maintain a strong CIA triangle while getting the most out of your AWS subscription.

Credential Management

Amazon provides several ways to manage user authentication via the Identity and Access Management service, or IAM for short. Certain methods are only available to certain services, however.

- Passwords

The most straightforward way to access AWS is with a simple username/password combination.

- Access Keys

The next way to access AWS is with a pair of keys which AWS calls the `AccessKey` and the `SecretKey`. This combination can be created, revoked, and deleted all from within the IAM service.

- Amazon CloudFront Key Pairs

Specifically for use with Amazon CloudFront, AWS can also generate a key pair for you to use.

- SSH Keys

Although SSH keys are also acceptable to remote into your EC2 instances, these are not generated through IAM. The SSH keys you can generate through IAM are specifically for use with the CodeCommit service. This is similar to a git repository and will secure who is allowed to push code into the repo.

- x.509 Certificates

AWS Credential Best Practices

- talk about utilizing different credentials for different roles, essentially RBAC for AWS

Specifying Credentials

- talk about how the -profile parameter can be used on any PowerShell command to tell the command what credential set to use

Using Multiple Named Profiles with AWS

What if you have multiple profiles? I run into this issue often, since I have multiple clients with multiple services and need multiple levels of access.

Configuring Federated Identity Providers

Managing the IAM module for all of the users in an organization doesn't always make sense, especially when your organization gets into the hundreds or thousands. Managing group membership would become a full time job by that point!

ADFS

SAML

Chapter 4: The big 3, IAM, EC2, S3

I wanted to dedicate a chapter to the "big 3" services in AWS, since these are the ones that are used in 99.9% of environments (which is a totally valid statistic that I definitely did not make up). At the very least, these are services that everyone will want to be familiar with, since each of them are a building block of the greater AWS ecosystem.

Managing IAM with PowerShell

Identity and Access Management is arguably the most important job that PowerShell does in it's normal lifecycle of Active Directory, but when we couple it with Amazon Web Services, it becomes even more powerful. It's hard to get too far in a PowerShell career without touching Active Directory at least once, and the same goes for AWS IAM. The Identity and Access Management resource determines who can access what resources, and when. By using PowerShell, we can ease the creation of user, group, and policy objects on the AWS platform, and a solid understanding of how to create, modify, and manage each object is critical to being able to automate the creation and manipulation of other AWS services.

IAM Objects

IAM Objects can be broken down to 3 types.

1) Users 2) Groups 3) Policies

Users are the lowest level object in AWS IAM. Users are individual accounts that can contain a unique permission set and have unique credentials to manage AWS resources. This means that users aren't necessarily a 1:1 relationship with an individual human being, but rather that they are a 1:1 relationship with a goal or an objective. For instance, as an Administrator I might manage the servers for 3 different projects. I wouldn't want to use the same set of credentials for all of the projects in case they have resources that shouldn't overlap. For instance, if there was client personal information stored on one server, I would want to organize my credentials so that I wouldn't even be able to perform a data transfer between the project with sensitive personal information and one with public information.

Groups are a way that we can help organize our users and automate a portion of our account management. By adding users to groups, you're able to apply the same set of permissions to individual user accounts in multiples instead of in the 1:1 relationship as you would need to at the User level. With groups, if you have a team that all works on a similar set of services, you can add each of the user accounts necessary to run and maintain that service into the appropriate group in order to set all of the permissions at once. If those permissions ever need to be modified, they only need to be modified once - at the group level.

Policies are permission objects that can be applied at a group or at a user level. <insert what all you can do with policies on AWS>

Creating IAM Users and Groups

The nice folks at Amazon had a great sense of forethought when they went to create the PowerShell module. The nomenclature across the board is very consistent, and it follows Microsoft best practices of using approved verbs for specific purposes. In this regard, you'll find that many of the upcoming code sections seem very repetitive, despite all doing very different things.

In order to create any IAM object on AWS, we'll use different permutations of the "`New-`" verb set of the PowerShell module.

```
PS> New-IAMGroup -GroupName "BookUsers"
```

By using the above command, you'll create a Group with the name of "BookUsers". The one parameter is the only parameter necessary in order for the command to work. There is a caveat, though, as you'll have to use alphanumeric characters (both upper and lower case are okay to use) and you can't use a space in the group name. There's a limited set of symbols you can use, as well (_+=`,.@-`). Now, let's create a User:

```
PS> New-IAMUser -UserName "Chapter.4"
```

Similarly to the last command, this command will create our new user, with the name of "Chapter.4". Just like in the Group creation command, the User creation command has a caveat in what you can use for the Users's name. Your UserName will need to be alphanumeric characters (both upper and lower case are okay to use) and you can't use a space in the username. You can use the same symbol set as you could with Groups (_`+-,.@-`). One other interesting note about usernames is that they are not case sensitive, so you can't create a user "Chapter.4" and "chapter.4".

Minimally the above commands will work, but before long you'll want more robust options for User and Group management.

Modifying IAM Users and Groups

Deleting IAM Users and Groups

Managing IAM Policies

IAM and CloudFormation

Best Practices

Monitoring

Security

Managing EC2 with PowerShell

Best Practices

Monitoring

Security

Managing S3 with PowerShell

Best Practices

Monitoring

Security

Chapter 5: Integrating more AWS Services

It’s simply impossible to talk at a deep level about all of the AWS services in a book anymore, not only because of the sheer number of services, or the complexity of each individuals network, but mostly because of how quickly the services are released and updated! There are now over 99 AWS services (at the time of this writing) and each of them has a dedicated team bringing new features and bug fixes to life every day. What we can do in this book, however, is go over each category of service at a high level, and touch on a few of the more commonly used or straightforward services that AWS has currently released.

Compute Resources

Storage Resources

DataBase Resources

Migration Resources

Management Tools

Machine Learning

Application Integration

Analytics

Chapter 6: AWS Best Practices

Talk about AWS Trusted Advisor?

- Cost Optomization
- Fault Tolerance
- Performance
- Security
- Service Limits

Access Management

Considerations:

- Network Accessibility
- Authentication
- Vuln scanning and patching

Trusted Advisor Concepts:

- Unrestricted Security Groups
- S3 Bucket Permissions
- EBS/RDS Public Snapshots
- Exposed Access Keys

Reliability

Considerations:

- High Availability?
- Data Backups
- Physical Locality?

Trusted Advisor Concepts:

- EBS Snapshot Age
- S3 Bucket Logging
- RDS Multi-AZ

Performance

Considerations:

- CPU Utilization
- Network Latency
- Disk IO
- Throughput?

Trusted Advisor Concepts:

Cost

Considerations:

- Infrastructure Scaling
- Software Licensing
- Database Mangaement
- Human Effort

Trusted Advisor Concepts:

- Auto-Scaling
- Underutilized Resources

Account Hygiene

Impacts of bad hygiene:

- Cost
- Security
- Too many choices

Taking Action to Remediate

Operational Reviews

- Allow you to identify usage patterns
- Communicate new requirements
- Assist teams in learning
- Measure Progress

Anatomy of a Review

- Communication of Account Updates
- Update users (team) information
- Trusted Advisor - Trending and Deep Dives
- Follow up from previous reviews

Review Automation

Use TrustedAdvisor API

Automated Remediation

Take known operational deficiencies and come up with “opt-in” automations for your users.

Low risk automations:

- Naming Conventions for server names

High risk automations:

- Public S3 buckets get shut down ASAP.

Amazon Well-Architected Reviews

Series of whitepapers that cover same pillars as trustedadvisor does.

Establish (enable tenants), Monitor(Measure and Report), Operate(Correct Errors and Issues), Optimize(Change and Improve).

Chapter 7: Security in AWS

Built-in Mitigations for Common Attacks

Compliance Program

Chapter 8: Billing

We’re going to talk about how to manage your bills.

How to manage budgets

How to monitor your spend

How to reduce your bill