# Introduction to AWS

March 2, 2021 by [Jawad Hasan Shani](https://hexquote.com/author/jawadhasan80/)

In this post, we will get a high-level introduction of Amazon Web Services (AWS). We will see few of its main services and we will also start setting up our development environment to build applications with AWS.

Amazon Web Services is a subsidiary of Amazon providing on-demand cloud computing platforms and APIs to individuals, companies, and governments, on a metered pay-as-you-go basis.

To start with AWS services, you will need an AWS account, which is very easy to setup and you can use free-tier for some of the upcoming demos. Free-tier is great when you are just starting with AWS. It allows you to try some AWS services free of charge and once you are comfortable with AWS, you can run bigger workloads and you will be invoiced accordingly. You can find more information about billing on official AWS website.

AWS is a collection of over 200+ services to choose from and develop with. AWS provides on-demand cloud computing platforms and APIs on a metered **pay-as-you-go** basis. Most services can operate independently or by integrating with other services. Few of the popular services offered by AWS are:

* EC2
* S3
* RDS
* DynomoDB
* ElasticCache
* AWS Lambda

Most services in AWS are accessed with a TCP connection. Often this might simply be HTTP.

**Each instance of service often called a resource**, is give a local IP address with which other services can communicate. This IP is not connected with outside world but you can assign an external IP, if needed.

Alternatively some services should be referenced by their ARN (Amazon Resource Name) instead of IP.

## AWS Tooling

Options available to us are:

* Web Console
* CLI
* SDKs

Even though most AWS can be worked with TCP connection, SDKs make this work even more easier. There are SDKs available for .NET, Java or other popular programming languages.

These SDKs provides robust APIs that enables you to interact with services and resources inside your codebase. SDK APIs are very similar in different languages.

In order to connect your local code to the cloud, we will need to use an AWS **access key** to properly configure the SDK to access our account in the cloud. AWS access-key gives access for SDK & CLI.

## Elastic Compute Cloud (EC2)

The core of the web of AWS is EC2. You can think of this as service that **provides servers** where your applications will be running.

EC2 also forms the basis of many other services as well, such as

* Relational Database Service (RDS).
* Elastic Bean Stack.
* etc.

Compute refers to compute or server resources:

* Application Server
* Web Server
* Database Server
* Mail Server
* Media Server
* File Server

In AWS they are called EC2 instances instead of servers.

Instances are **pay as you go**. You only pay for running instances and only for the time they are running. EC2 instances comes with some built-in options:

* Broad selection of HW/SW
* Global Hosting

## Identity and Access Management (IAM)

* Configure and manage users.
* This service is integral part of AWS.

## Cloud Watch

* Used for **alarms**and **logging**.
* Also integrated very deeply in many AWS services.
* With cloud watch you can configure alarms to notify you or trigger automatic activity if service go down, overloaded or acting crazy.
* Cloud watch can be used with SNS to send alerts to end-points (e.g. email, SMS).

## AWS Lambda

AWS Lambda is a compute service that runs your code in response to events, for example, when **data is inserted in a DynamoDB** table or when a **file is uploaded to S3** or whenever a **request hits one of the API endpoints** and automatically manage the compute resources for you, making it easy to build applications that respond quickly to new information.

**Lambda Core Features**

* Runs code on demand.
* No Servers: Servers, OS Maintenance, Scaling, Provisioning and Deployment is managed for you.
* Code starts running within milliseconds of receiving the event trigger.

## Simple Storage Service (S3)

Amazon S3 is a storage service that can also function as a web server.

In S3, you put your files in buckets.

## CloudFront

CloudFront is a Content Delivery Network (CDN) service that will copy your app to edge locations around the world to improve the speed in which your app is served to your users.

## API Gateway

API Gateway is a service that allows creating a REST API fully managed by AWS that acts as the front-end for other services.

It makes it easy for developers to publish, maintain, monitor and secure APIs at any scale. In other words, it allows you to create API endpoints for your backend application without having to manage the backend servers themselves.

Also, when we use API Gateway, we did not just built a single system with finite resources, but a fault tolerant scalable system behind the scenes to support and grow our APIs.

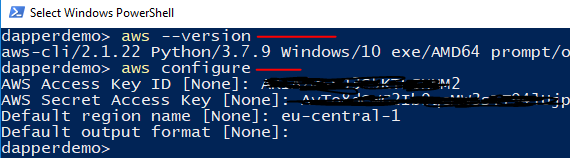
## Install AWS Command Line

I am using windows and for other OS you can refer to AWS website for instructions. Download installer from AWS official website and [install](https://docs.aws.amazon.com/cli/latest/userguide/cli-chap-install.html)CLI. After install you can check the installation by running following command:

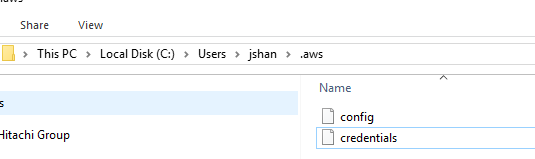
aws --version

Next, we configure aws by using the credentials we got earlier:

aws configure

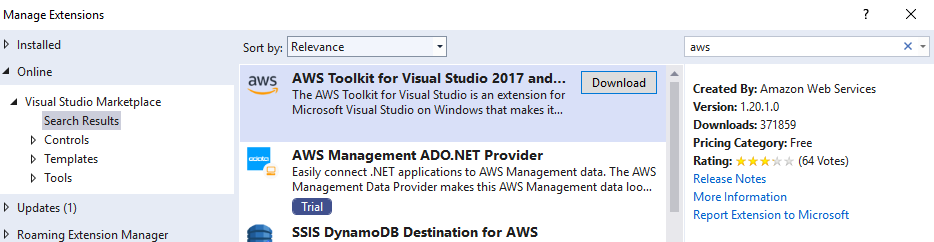


This will ask you to provide user access-key-id and secret-key and you can enter those details. Once its completed, you will see following files on your machine:

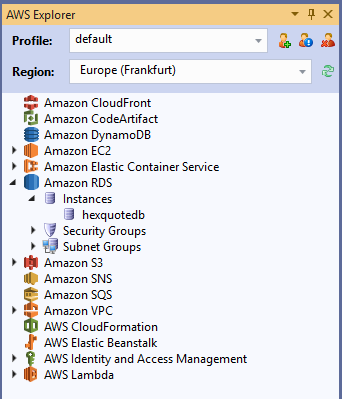


## Install AWS Toolkit for Visual Studio

I will be using Visual Studio for my development purposes. You can install the tool-kit from within visual studio extension menu item:



After installing the extension, you can open AWS explorer inside visual studio and access AWS Resources:



## Summary

In this post, we covered few of the AWS resources and we also started setting up our development environment.

AWS Relational Database Service (RDS) – PostgreSQL in Cloud

March 6, 2021 by [Jawad Hasan Shani](https://hexquote.com/author/jawadhasan80/)

Introduction

In my previous post “[Introduction to AWS](https://hexquote.com/introduction-to-aws/)“, I gave a high level overview about various amazon services and their common uses.

Today, we will go into details of Amazon RDS. We also setup a PostgreSQL instance using this service and connect to it using a tool Azure Data Studio

Amazon RDS is a web service that makes it easier to setup, operate and scale a relational database in the cloud.

This service can help free up time so that it can be refocused to creating business solutions instead of managing hardware and database installations.

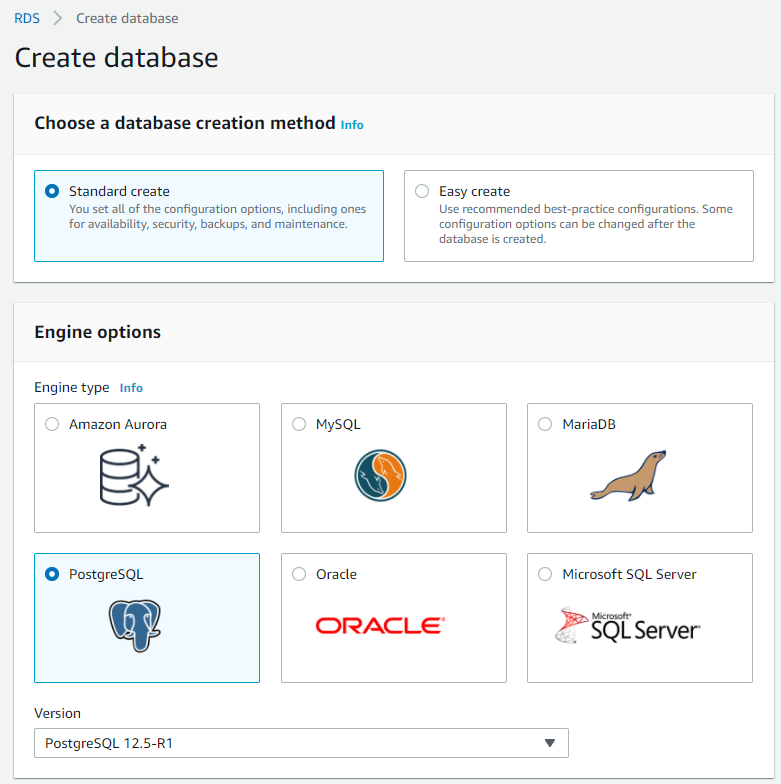
You can use AWS RDS with variety of engine e.g. SQL Server, PostgreSQL, MySQL, MariaDB, Oracle and Aurora. There are some settings which are common to all the engines and some which are more specific to a particular database. In this post we will also see how to work with these settings (actually AWS web console makes it very easy to get it setup).

Some of the supporting features of RDS are:

* Read Replicas
* Database Backup (Snapshot)
* Database Monitoring
* others….

Creating an RDS DB Instance

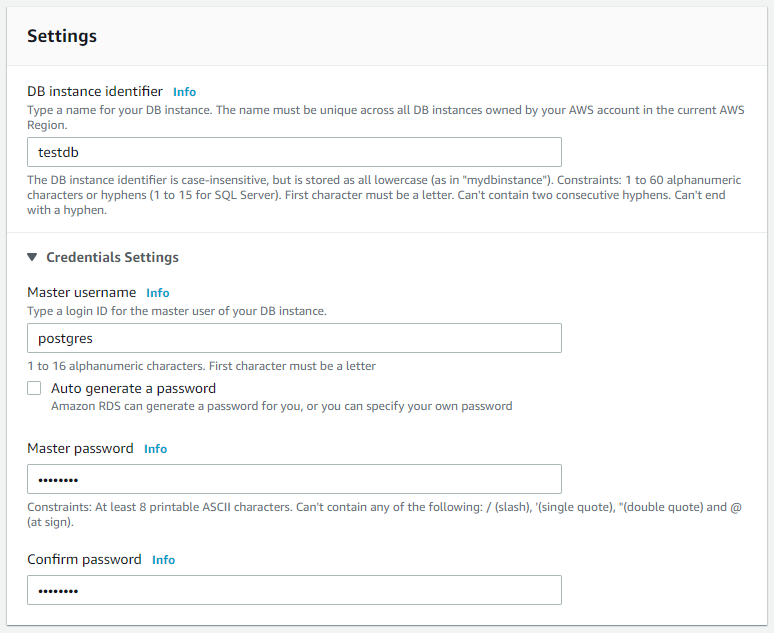
From your AWS Web Console you can select **RDS** and choose **Standard** **Create**and select a database engine in this case PostgreSQL:



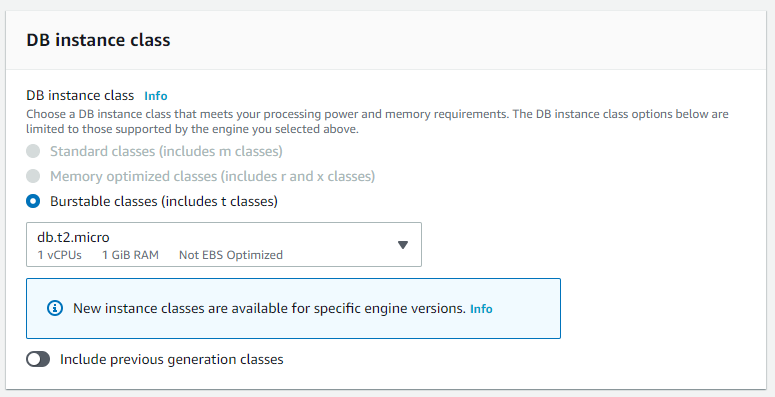
Next, we can choose a template to meet our requirement. We will select **Free tier**:



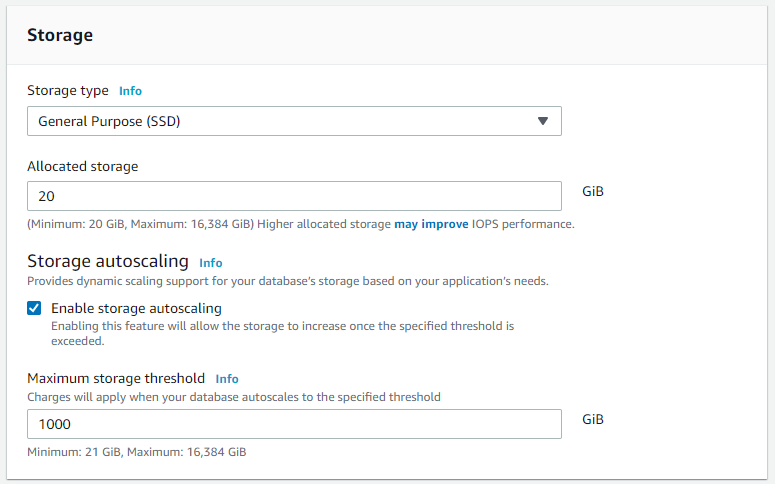
Next, we will choose a DB instance name, username and password. These settings are same of other database engines as well:



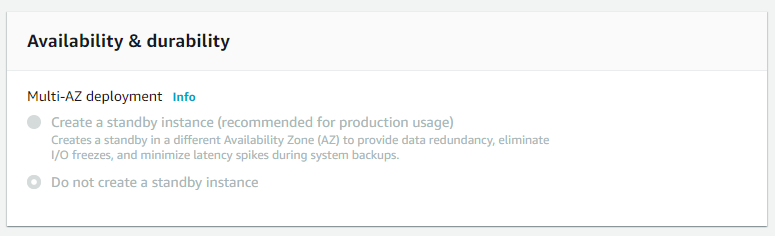
Next, we can select DB instance class, I am selecting here a very basic one (free tier) which is ok for testing or small PoC application:



For Storage, setting, we can go with the defaults here:



Multi-AZ means having double all the resources. You can see that Multi-AZ is grayed out (this is due to free tier. Selecting this option will result in almost double the cost but it might be a requirement for production scenarios.



We selected free tier in the start, that’s why most of the options are preselected for us as well.

But decisions about choices above normally comes down to a balance of performance needs and budget, both of which may be unknown at this point in the time. That is one of the benefits of using a **Platform as a service** for your database or any hosted solution for that matter. These choices can be always changed later without significant downtime.

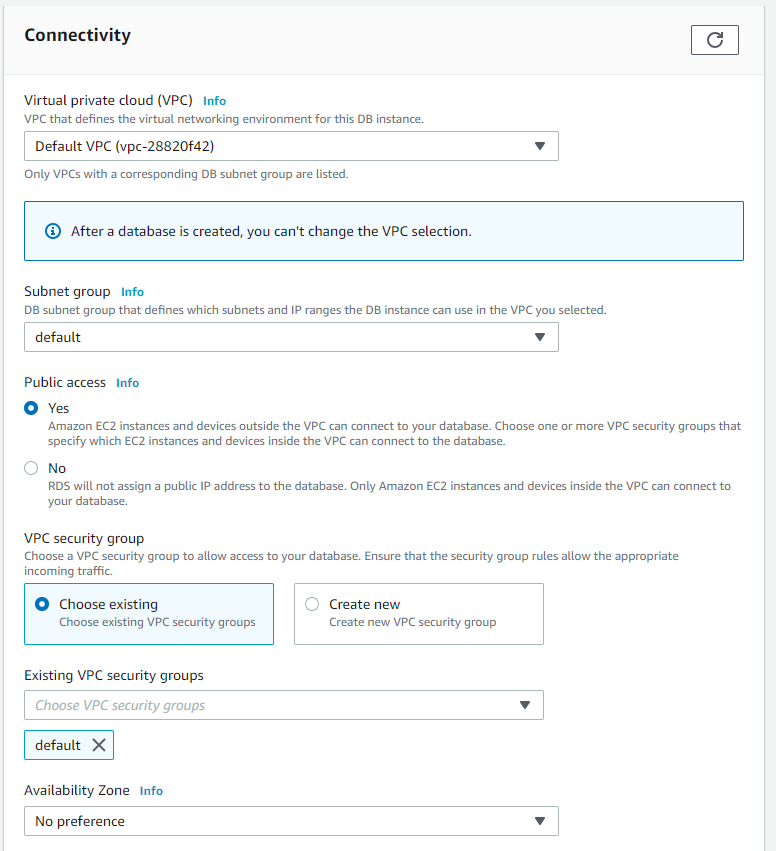
Networking

Next set of configurations is for VPC (Virtual Private Cloud), that this database instance will be accessible from. Defaults here are acceptable for testing purpose, but you can modify those as per your situation.

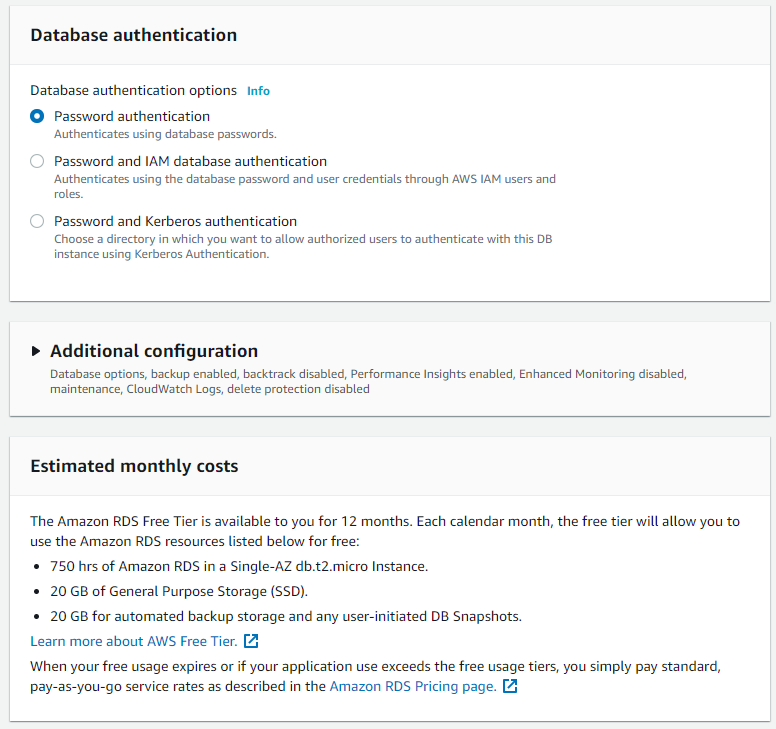
(e.g., you may have a scenario with an EC2 instance that is hosting your application. Instead of exposing the database instance through a public network, you could add both resources to the same VPC so that the application can communicate with the database over a private network.)

Notice, that I’ve selected **yes** for **Public access**. This option allow public accessibility to this db instance.

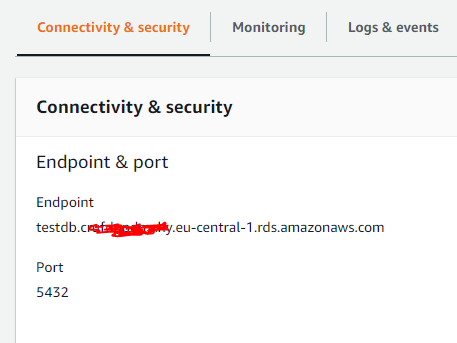
For **Security groups** setting, we have selected **default**. Security groups are part of the VPC that identify the network traffic rules for inbound and outbound traffic. So for **public accessibility** selection we made is going to be configured within this security group.



For database authentication, default Password authentication is ok for us.

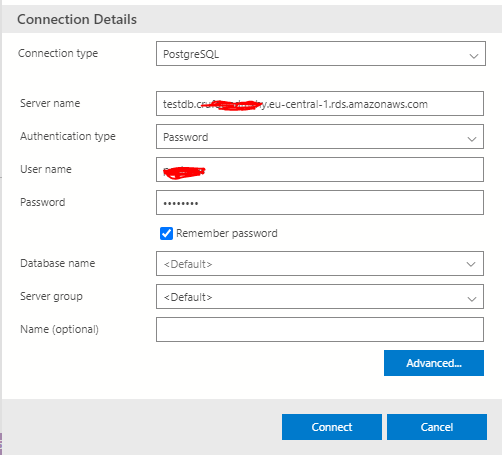


We can click **Create Database button** at the end of the settings and in few minutes database instance will be running and ready to use. You can see the details of that instance as shown below and I will use this information to connect to it from Azure Data Studio running on my windows machine:

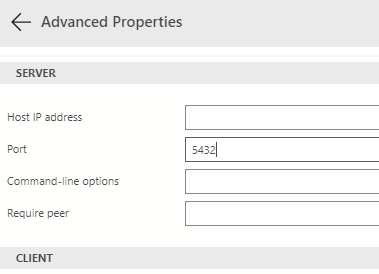


Connecting with Azure Data Studio

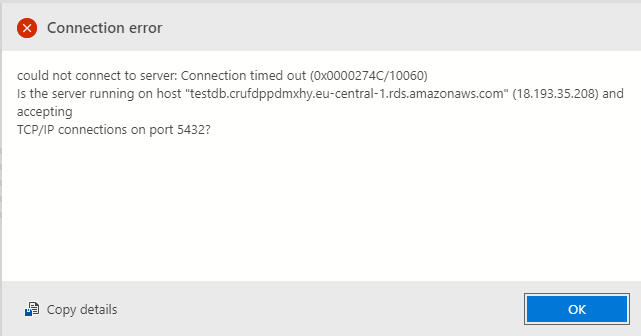
Following is the connection dialog in Azure Data Studio for the PostgreSQL instance:



Click **Advance**to enter Port number:

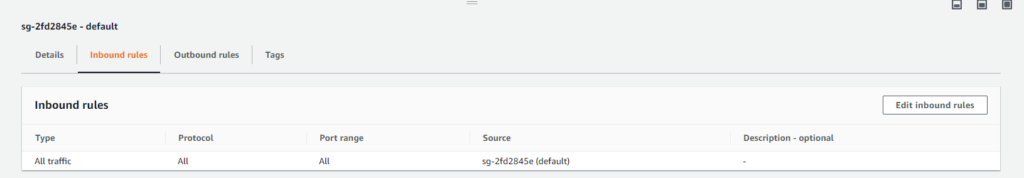


and Click Connect to make a connection with the db instance. You may see a connection error dialog similar to following:

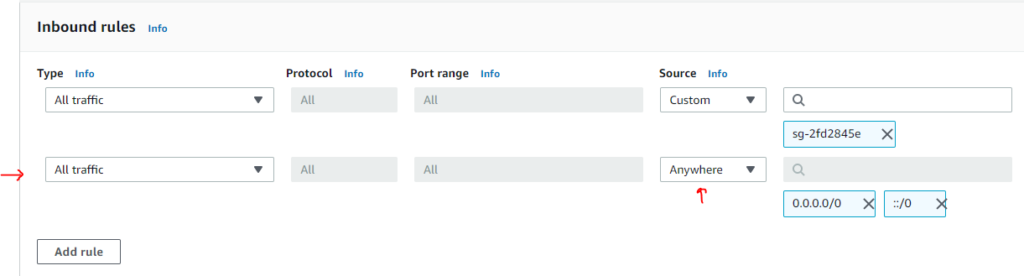


We need to configure **Inbound rules** for **Security Group** to allow incoming traffic for our db instance.

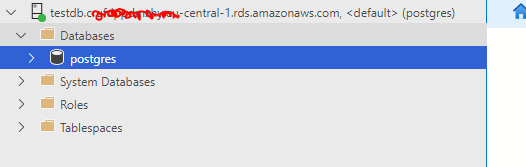
You can go into **security group** details and **Edit Inbound rules** from following screen:



Add a new Rules to allow incoming traffic from **Anywhere** (not a production recommendation) as shown below:



With the inbound rules in place, try to connect again and this time, we are connected successfully:



Now you can start writing some SQL Queries, create databases, connect applications to it etc. You can also check my other article [Fun with SQL using Postgres and Azure Data Studio](https://hexquote.com/fun-with-sql-using-postgres-and-azure-data-studio/).

Summary

Amazon RDS makes it very easy to spin up databases in cloud and saves you a lot of time by managing a lot of infrastructure and database setup concerns for you. Its easy to start with and you can also customize various settings as per your application requirements.

Amazon VPC Basics

March 16, 2021 by [Jawad Hasan Shani](https://hexquote.com/author/jawadhasan80/)

Introduction

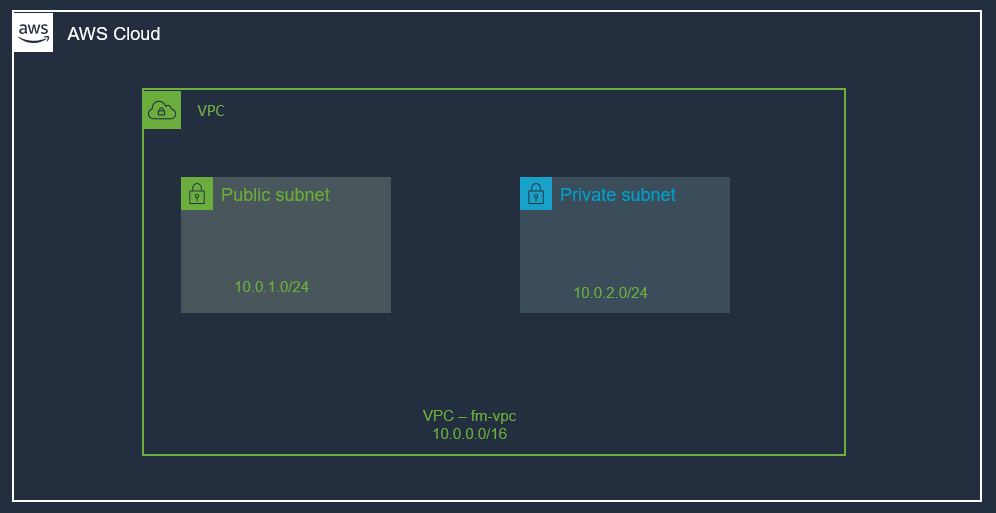
Amazon Virtual Private Cloud (Amazon VPC) is a service that lets you launch AWS resources in a logically isolated virtual network that you define. You have complete control over your virtual networking environment, including selection of your own IP address range, creation of subnets, and configuration of route tables and network gateways.

A default VPC is created automatically by amazon inside each region for your account. This way, you don’t have to worry about it when starting with AWS services.

However, typically you will create your own VPC and understanding of it is very important for moving forward with AWS.

In this post, we will learn about VPC, Subnets and some of the components like CIDR blocks etc. We will create two subnets: one public subnet which we will configure later to connect with internet via an internet gateway and one private subnet which will have no direct connection with internet. We will be using AWS web console to setup this VPC.

At the end of this post, we will have the following infrastructure setup:



Benefits of Using Amazon Virtual Private Cloud (Amazon VPC)

When you are creating a VPC, you are actually setting up a data center in cloud, utilizing amazing AWS infrastructure, its secure, highly available and scalable.

* Secure and monitored network connections.
* Simple set-up and use.
* Customizable virtual network.

VPC Highlights

* A VPC is a logically isolated piece of AWS cloud dedicated to your company. Means, you can run applications on overly provisioned, highly available and redundant infrastructure setup and it is managed by AWS. All the complexity of setting up a data center with cables, server racks, hardware, power supply etc. all are managed by AWS.
* A VPC belongs to a region.
* A VPC spans all availability zones.
* You can have multiple VPCs per region.
* VPC contains one or more subnets.
* A Subnet is tied to a single availability zone.
* EC2 instances launch into subnets

IP Addresses

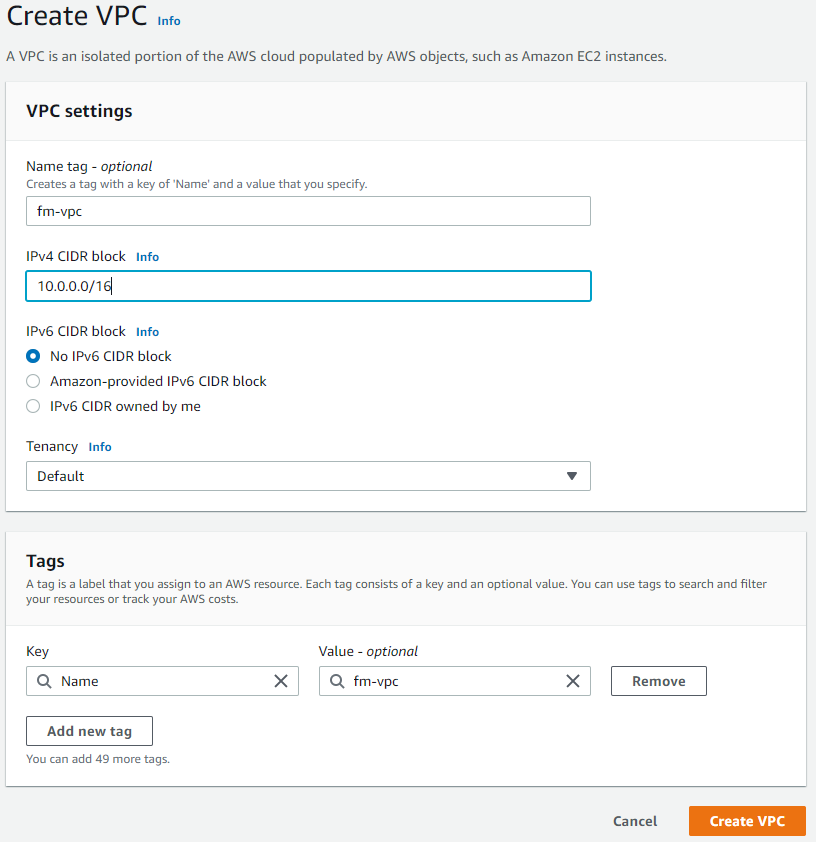
* A big part of your VPC are IP address.
* CIDR (Classless Inter Domain Routing) blocks are used for these IP address. These IP addresses are private and are not publicly routable.
* Initially you set a CIDR block for VPC. Then you assign portions of this CIDER block to individual subnets inside the VPC.
* CIDR blocks (Sizes between /16 and /28 are allowed)
  + 10.0.0.0/16 (65,536 possible addresses) –> 10.0.0.0 – 10.0.255.255)
  + 10.0.0.0/28 (16 possible addresses) –> 10.0.0.0 – 10.0.0.15

One more very important thing to remember about VPC CIRD block, is that, you should not have overlapping CIDR blocks, even in a comparison to your on-prem data center. Treat your VPC as your cooperate data center. If you are not careful then you will have issues in future when you try to do some thing advance with networking e.g. setting up VPN connection or likes. So, give yourself a favor, do not use overlapping CIRD ranges from day one.

There are Public IPs and Elastic IPs available for us as well to use. If you need a fixed static IP for example for an EC2 Instance, you can allocate one Elastic-IP and use it. More on those in next post.

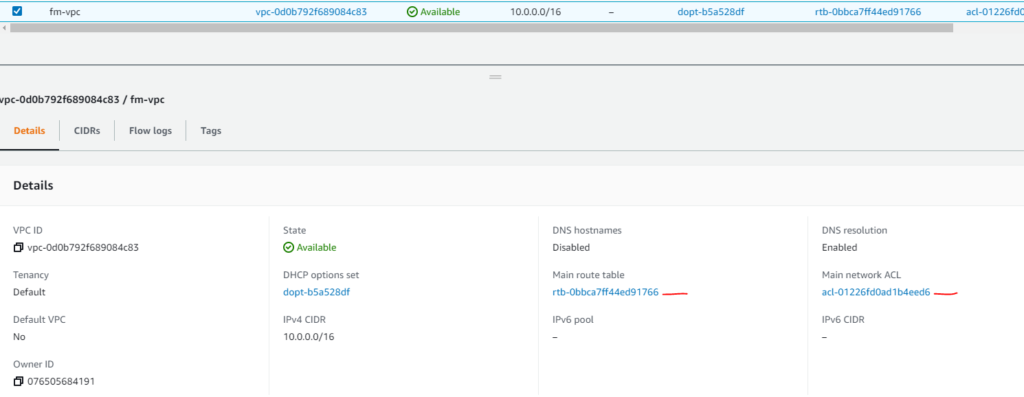
Create a VPC

We will create a VPC from web Console. We need to provide a name and IPv4 CIDR block for the VPC as shown below (/16 means that first two octets are for network address):



You can also check **[cidr.xyz](https://cidr.xyz/" \t "_blank)** to see a visualizer tool for CIDR blocks.

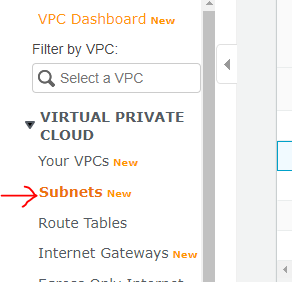
The following picture shows the created VPC. AWS also created some other items automatically for the VPC e.g. **Main route table** and **Main Network ACL** (more on those later):



Subnets

We can segment one VPC into subnets using web console:

VPC –> Subnets –> Create Subnet



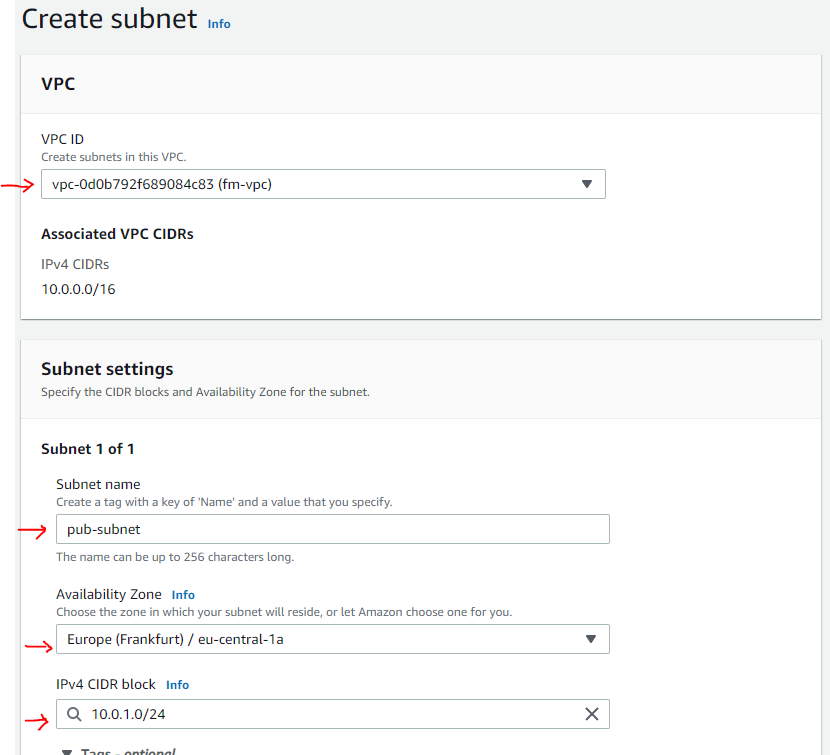
Create a Public Subnet

We need to select a VPC for the subnet, a name, CIDR block etc as shown below.

VPC CIDR info shown tells us what the available IP range is for the entire network, so that we can specify only a piece of that for this subnet. Note the CIDR block for subnet. When instances are launched, their IP address is determined by subnet CIDR.

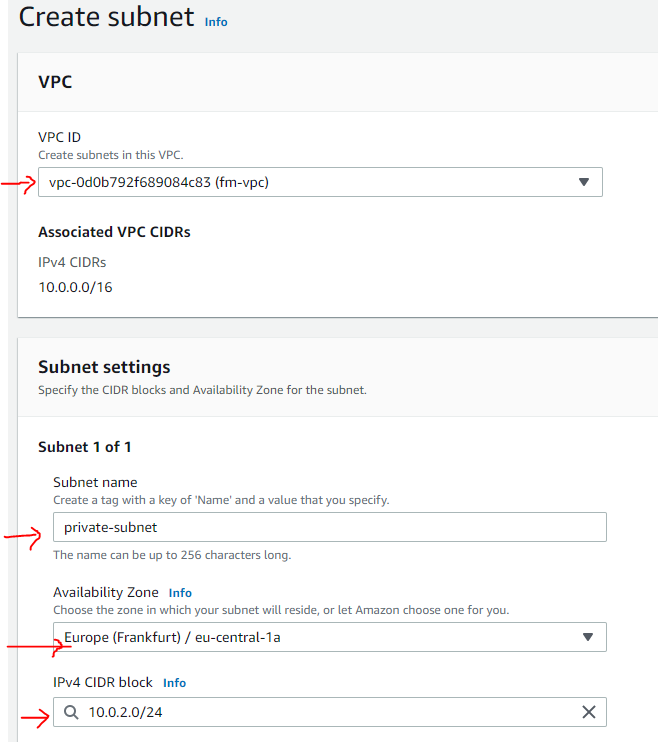
We are naming this subnet a public subnet, it will have internet access later and this can be used for jump box (**bastian server**) etc. (Bastian server can be used by devs to connect to VMs inside the VPC. We will need a public-IP as well for this purpose. The public IP address can be used within the scope of the VPC. More on this in another post).

For each subnet block, AWS reserves the first 4 and last address of each subnet.

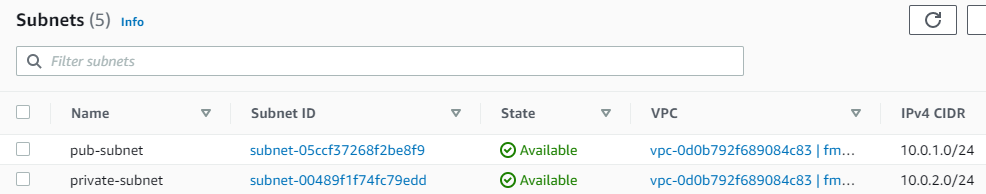


Create a Private Subnet

Similar to above, we will create a private subnet. One difference is that this subnet will not have direct internet access:



Subnets Created



Subnets for Application Tiers

If we want, we can create more subnets and use those for different tiers of our application e.g.

* –> api-myApp-subnet (10.0.3.0/244) — private subnet
* –> db-myApp-subnet (10.0.4.0/244) — private subnet

However, to keep things simple, We will have only two subnets (public and private) and we will not be creating additional subnets for application tiers.

Question

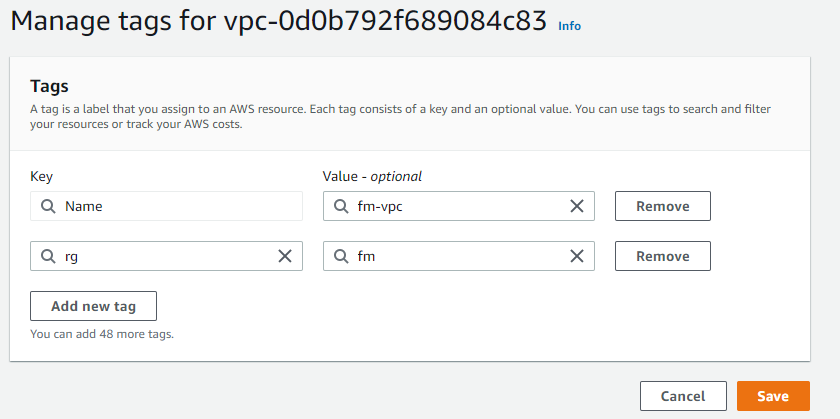
Isn’t this IP-Range going to overlap with another customer?

Answer

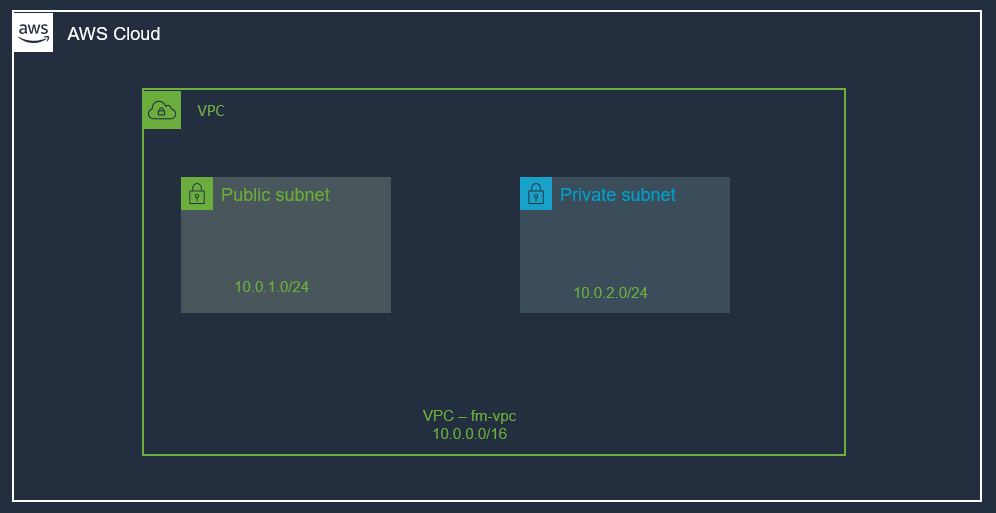
Absolutely Not. These VPCs are tied to your AWS account, therefore, it allows for (because this is a software-defined-networking) for each individual AWS account to leverage the same IP-range within the same region.

Naming your VPC

It will help you a lot to name your resources in AWS. You can also additional tags e.g rg –> Resource Group. Giving every single resource this tag, will help you managing your resources:



At this point, we have created our VPC with two subnets as shown below:



Summary

In this post, we learnt the basics of VPC and saw how easy it is to setup and how much time and effort can be saved using this instead of setting up our own data center.

We setup one VPC with two subnets. Right now, this VPC is not doing much and even we created a public subnet and a private subnet, they both are empty. No workload is running. Public subnet is only public in its name, in reality, currently, it is not different from private subnet (meaning it does not have a connection to internet).

# AWS Internet Gateway and VPC Routing

March 19, 2021 by [Jawad Hasan Shani](https://hexquote.com/author/jawadhasan80/)

## Introduction

In the previous post on [AWS VPC Basics](https://hexquote.com/amazon-vpc-basics/), we learned about VPC basics and we also setup a VPC with public and private subnets.

In this post, we will learn about another powerful component from AWS, the gift of internet, The Internet Gateway. We will also learn how routing works within VPC, how to setup route to internet gateway and our public subnet. This setup is very common for most of the applications on AWS.

An **internet gateway** is a service that allows for internet traffic to actually enter into a **VPC**. Otherwise, a VPC is completely segmented off and then the only way to get to it is potentially through a VPN connection rather than through internet connection.

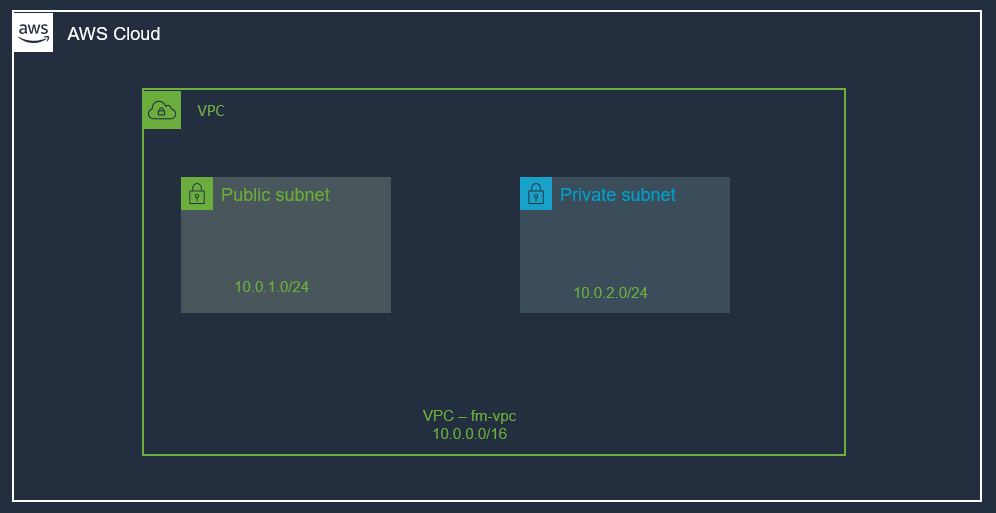
An Internet Gateway is a logical connection between an AWS VPC and the Internet. It is not a physical device. Each VPC has only one Internet Gateway. If a VPC doesn’t have an Internet Gateway, then the resources cannot be accessed from the Internet. Conversely, resources within your VPC need an Internet Gateway to access the Internet.

In later posts, when we will be creating EC2 instances in subnets, then we will see how it is used to allow internet access.

Also, In order for us to allow the developers to connect to the VMs, to a **jump server** or a **bastian-host**, we have to have this **internet gateway** first. We typically use a NAT instance or NAT Gateway for this purpose.

A Network Address Translation (NAT) enable instances in a private subnet to connect to the Internet, but prevent the Internet from initiating a connection with those instances. To do that, NAT maps all the private IP addresses assigned to the instances in the subnet to one public IPv4 address called the Elastic IP address.

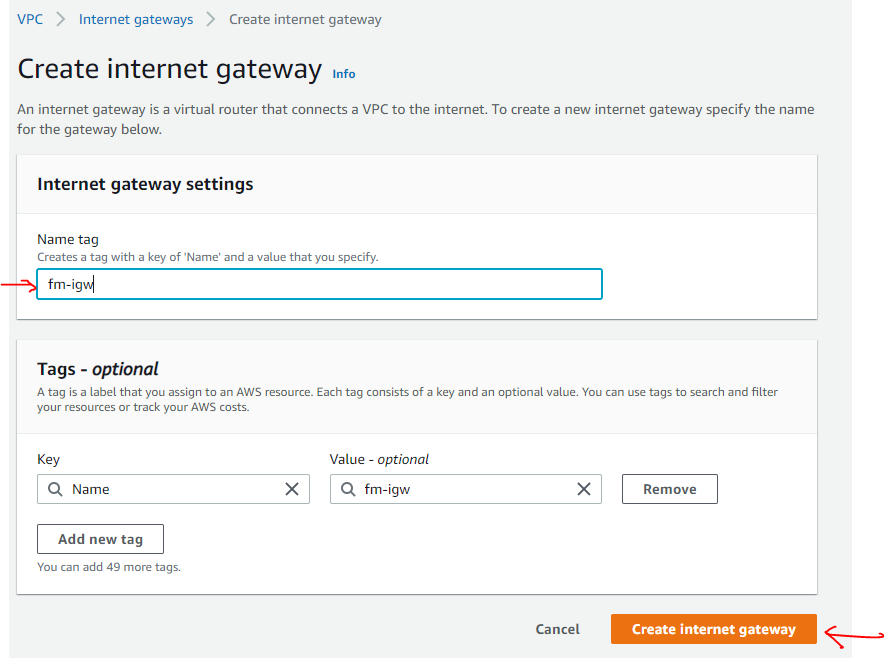
Here is how our setup looks like from the previous post:



In order to **allow internet traffic to our VPC**, we can use an **Internet Gateway**. Let’s create an internet gateway next.

## Create an Internet Gateway

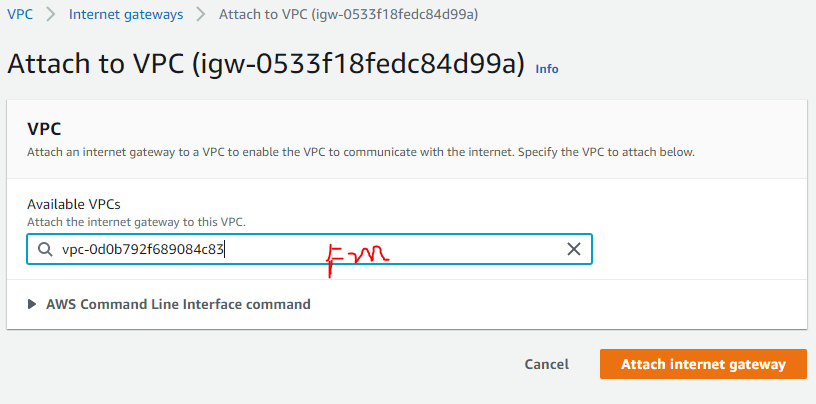
You can create an internet gateway by navigating to VPC and creating it via the console:



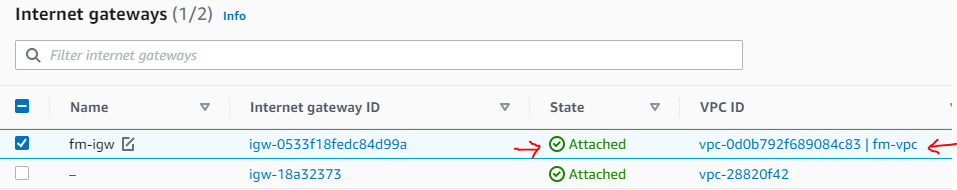
So, we created an internet gateway, but its just sitting there by itself. Its not doing anything yet.

Remember, internet gateway is used to **allow internet traffic to VPC**, so it needs to be associated with a VPC.

Select **Internet Gateway** Resource --> Click on **Actions** --> **Attach VPC** (select VPC created earlier)



Here is attached internet gateway:



This will allow traffic to enter the VPC, but we can lock it down. That’s where **Security Groups** are going to come into play (we will talk about those a little bit later).

Before we continue with anything else, lets take a step back and talk about **Route Tables**.

## Router and Route Tables

Every VPC has an implied router, you don’t need to create it, its already there. It routes the traffic within the VPC and then we also have route tables which contains routing entries. Normally we just work with route table entries. Think of a road and road-signs analogy. The router is the road and road-signs are the route table entries.

A route table specifies how packets are forwarded between subnets within your VPC, the internet, and your VPN connection.

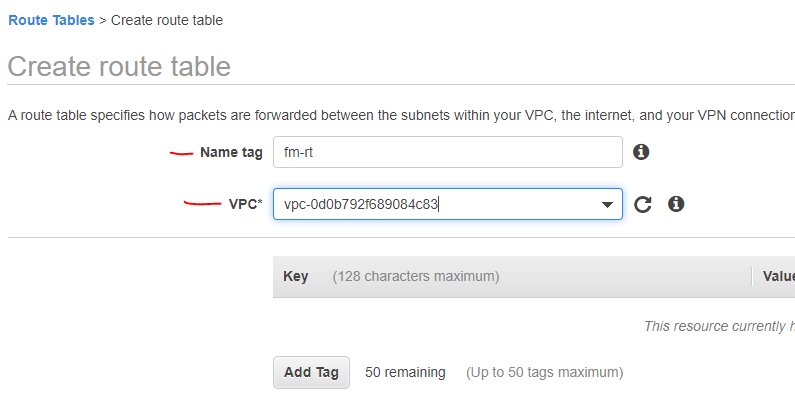
A route table contains a set of rules called routes which determine where traffic has to be directed. You can create as many route tables in a VPC as you want. Route tables act at the subnet level, not the VPC level. A route table can be associated to one or several subnets. By default, all route tables in a VPC have a local route for communication within the VPC. You can add custom routes in a route table by creating a new route defining which traffic (IP destination) must go where (target).

**If a subnet is associated to a route table redirecting all traffic to an internet gateway, it is called a public subnet.**

If you remember from the previous post, a primary route table was created automatically by AWS when we created our VPC. If you now visit the route tables page in AWS console, you will find a list of existing route tables. We can also create new route tables, and this is what we will do, we will create a custom route table as shown below:

### Create a Custom Route Table

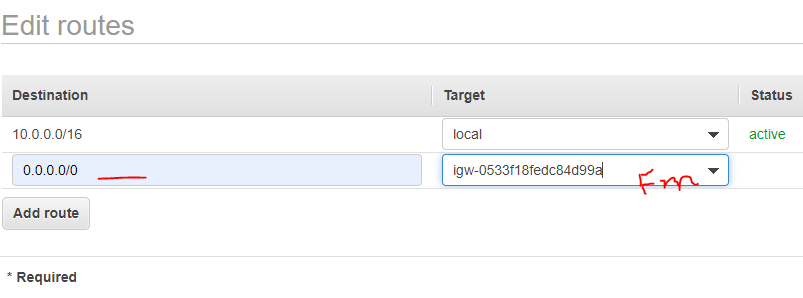
From the VPC page, select the Route Tables item and Click Create Route Table. We need to provide a **name** for route table and select our VPC as shown below..



Next, we will **add a route for internet gateway** to this custom route table.

### Add Route for Internet Gateway to custom route table

Select the route table, and Edit the routes as shown below:

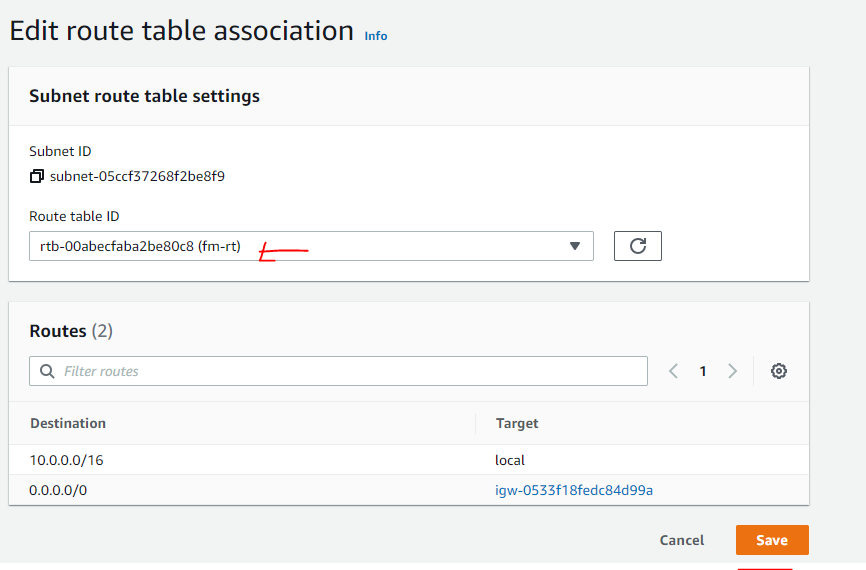


Notice that first entry (10.0.0.0/16) is for VPC local traffic and we added a catch-all route (0.0.0.0/0) and set its target to our Internet Gateway, which we created in the beginning of this post.

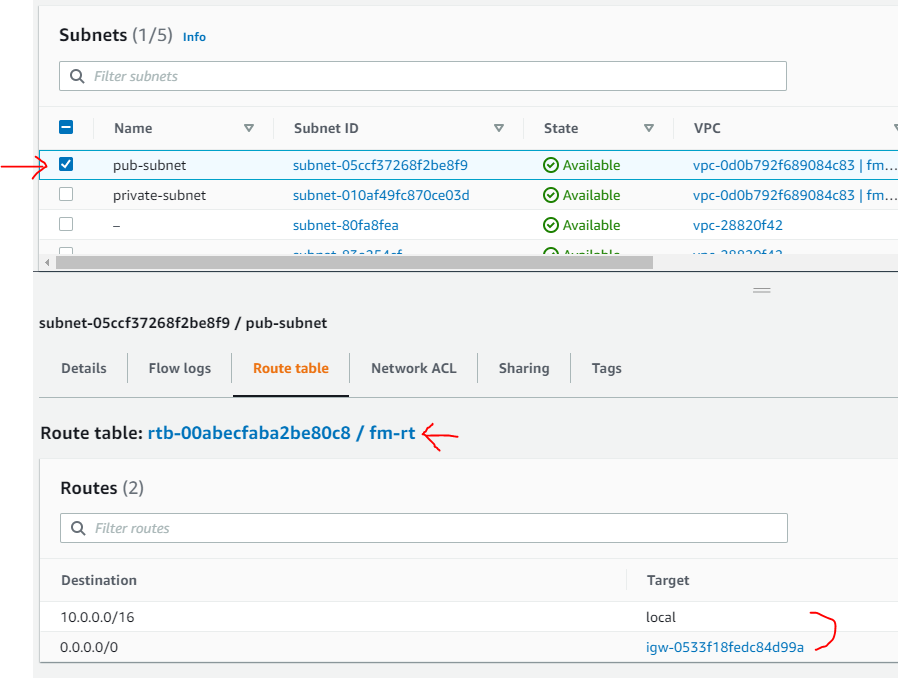
### Associate Custom Route Table to Public Subnet (for internet)

Our custom route table now have route with Internet Gateway. Next, we will associate it with the public subnet.

VPC–> Subnets –>Select **Public Subnet**–>RouteTables –> Edit Route Tables Association



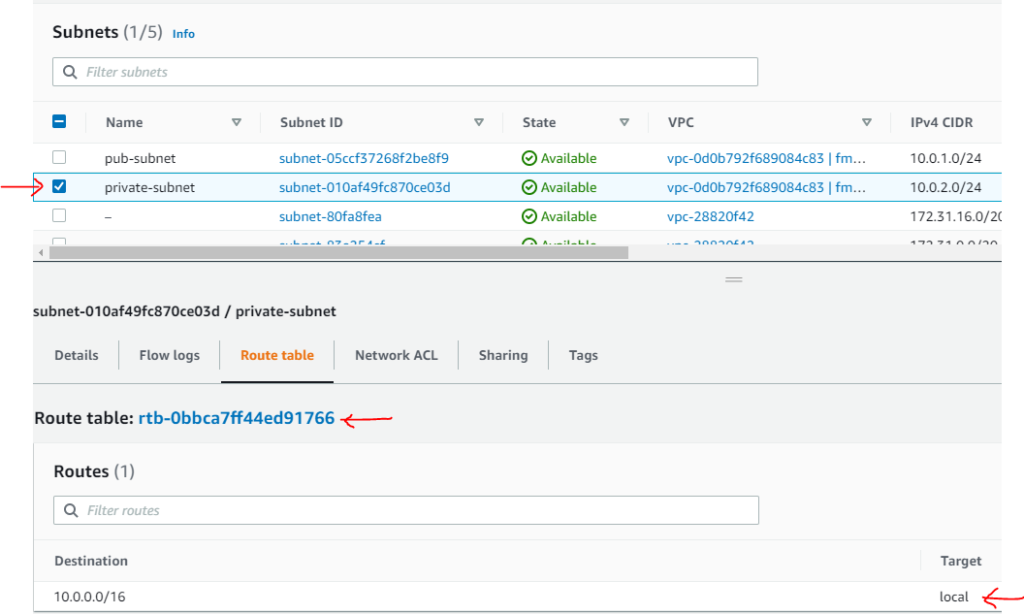
The following screenshot shows the association of route table with public subnet.



### Private Subnet Route Table Association

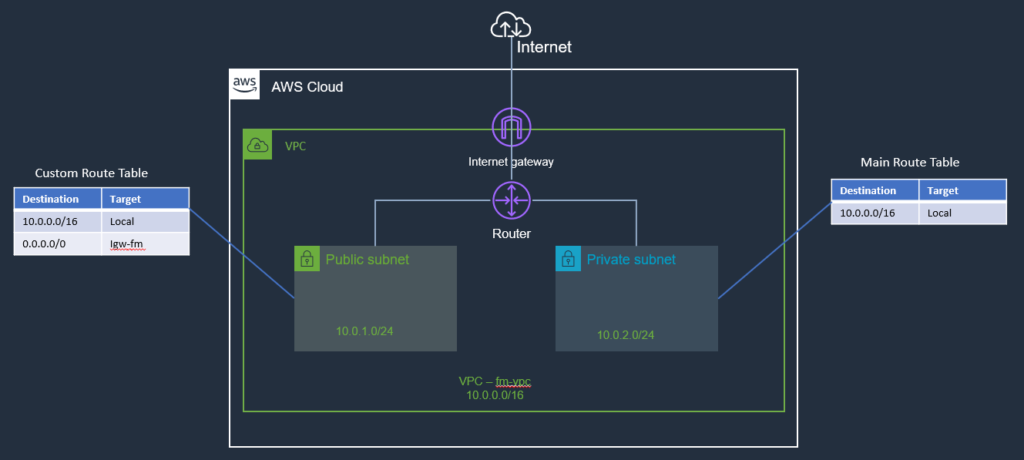
Our Private subnet is still associated with main route table (no internet) and that is what we want for our private subnet. So, we do not need to do anything.

**A private subnet is a subnet with no direct route to an Internet Gateway (IGW)**. In other words, it has no public IP addresses, no publicly accessible resources, no direct internet access to/or from that subnet.



## Infrastructure View

Let’s see how our infrastructure looks like now. The following picture shows these components:



## Summary

In this post, we continued our journey from where we left it in the previous post on [AWS VPC Basics](https://hexquote.com/amazon-vpc-basics/). We have extended our infrastructure a little bit by adding an internet gateway and then setup the routing by creating a custom routing table associated with **public subnet**.

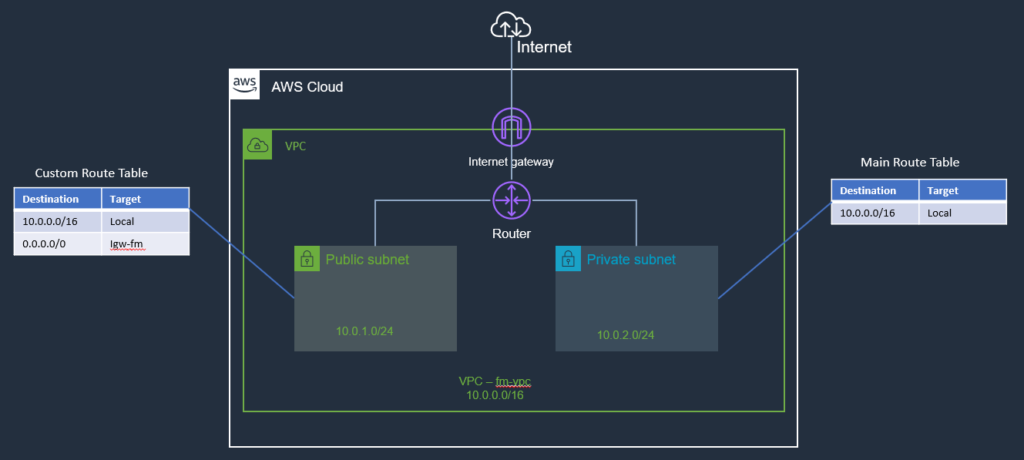
AWS Elastic Compute Cloud (EC2) Basics

March 24, 2021 by [Jawad Hasan Shani](https://hexquote.com/author/jawadhasan80/)

Introduction

In my earlier post on [AWS Basics](https://hexquote.com/introduction-to-aws/), we learnt that the core of the AWS is EC2. You can think of this as service that **provides servers** where your applications will be running.

We then started setting up our VPC, Subnets, Routing Table and we also setup and Internet Gateway and we have following architecture in place (you can check my earlier post on [AWS Internet Gateway and VPC Routing](https://hexquote.com/aws-internet-gateway-and-vpc-routing/) for details):



Now in this post, we will launch EC2 instances. We will start simple and initially **launch an Ubuntu instance in our public subnet** and another in **private subnet**.

We will learn about IP Addresses and also connect to public EC2 instance externally using SSH. But before we actually launch our EC2 instance, lets have a look on public and private IP behavior first.

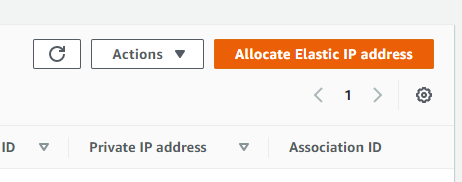
Public/Private IP Behavior

* When instances are launched, their IP address is determined by corresponding subnet CIDR.
* Private IPs:
  + Private IPs can not be addressed directly from outside (e.g. internet).
* External (public) IPs:
  + External IPs can be addressed directly from outside internet.
  + External (public) IP addresses are not assigned by default. You must specify you want a public IP on instance creation. You must either have auto-assign IP address enabled or explicitly check that in order to have an external IP address assigned to your instance.
  + External (public) IPs are assigned from AWS pool of IPs.
  + If you **start**, **stop**, **restart** an instance, this external IP can and usually does change.
  + If you want to keep the same IP address for instance during restarts, you can use **Elastic IP**.

Elastic IP

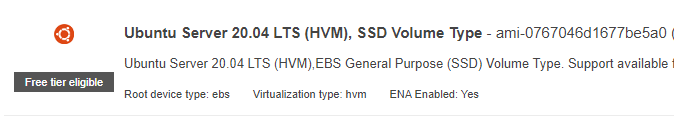
Elastic IPs are basically public IP addresses that are created, destroyed and assigned independently and can be associated with instances.

As I mentioned before that we will connect to the EC2 instance externally, we can create an Elastic-IP in advance and then use it later during the instance creation process. You can create Elastic-IP from the VPC dashboard as shown below:

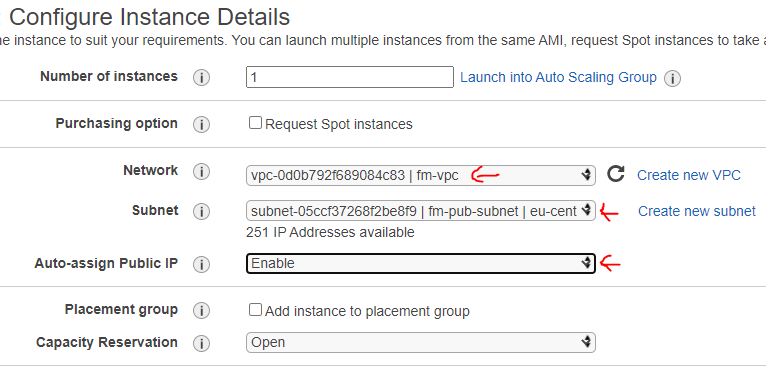


Launching an EC2 Instance

AWS console make this process very simple. Jump to the AWS EC2 console and provision a Linux server (Ubuntu) by clicking **Launch** **Instance** button to start the wizard. I will use Ubuntu Server 20.04 LTS (t2.micro type) as shown below:

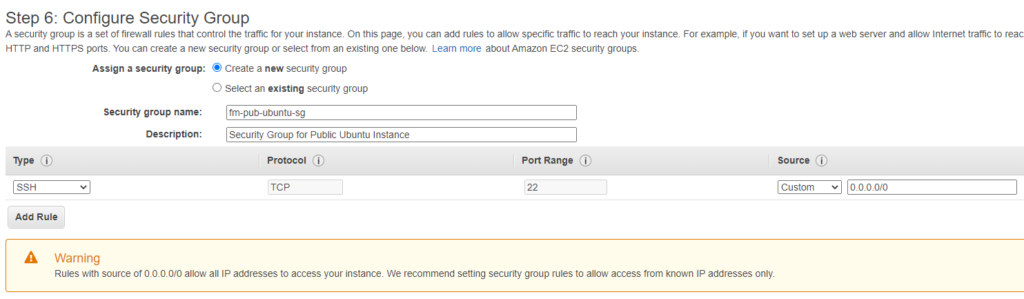


On Configure Instance Details screen, Select the **VPC**, I selected the **public subnet** and also notice **Auto-assign Public IP** is **enabled**:



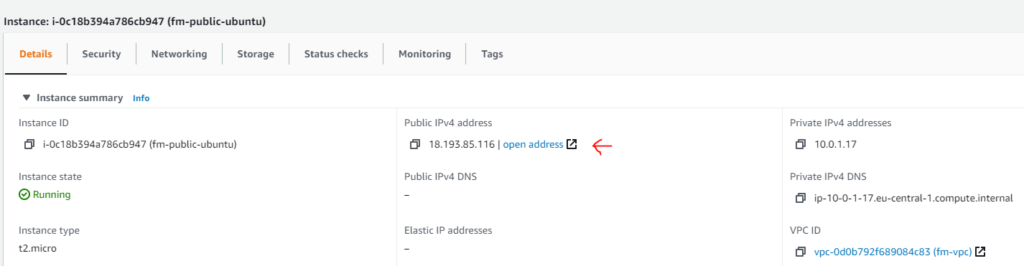
Other defaults values are fine for now. Click through other pages of the wizard, give the instance a**Name Tag.**

On **Configure Security Group** Page, There is already one SSH rule populated (we will cover security groups in details in later post). For now, accept this default rule:

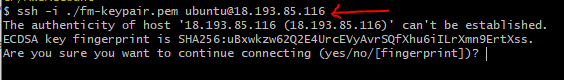


This rule is saying accept incoming SSH traffic from anywhere. Typically you should lock it down to may a single IP address or the likes. For this post, I will let this traffic come in from any source.

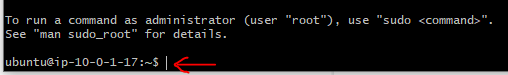
Click **Review and Launch**. It will also ask you to create or use a keypair. Keypair will be needed when we will SSH to this instance. In a few minutes, EC2 instance will be launched and show up in the EC2 Dashboard:



We can see that it has a public IP address. We also allow SSH traffic via port 22 in the security group. Let’s try to connect to it. I will be using Git Bash on Windows, you can use Putty or similar tool if you like.



and in a moment, we will be connected to the EC2 instance as shown below (notice that prompt is now showing the private IP address of EC2 instance as well).

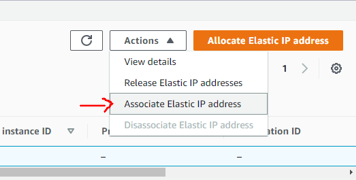


That was easy. One question, you may ask, If we are able to use the public IP the **why we created the Elastic IP** earlier in this post?

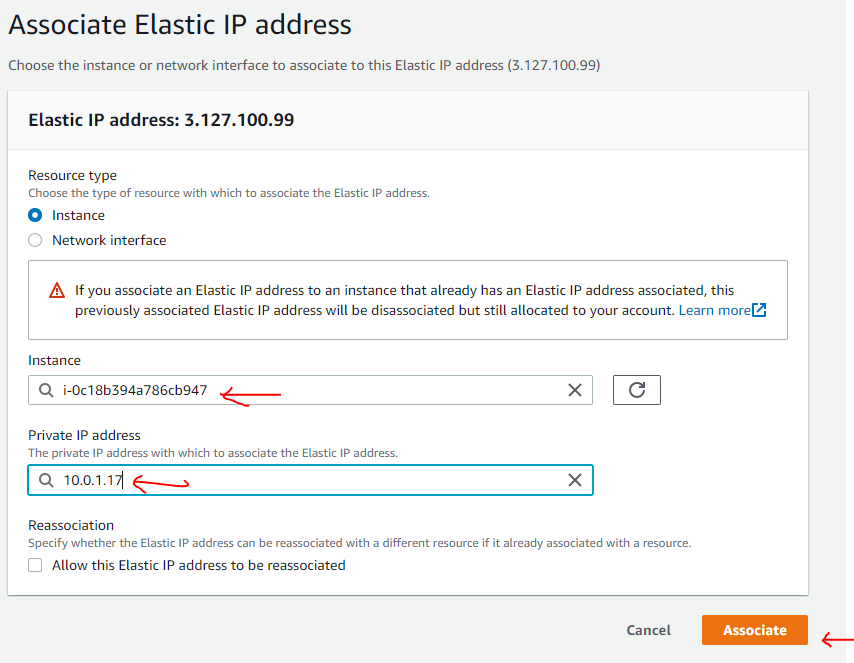
The reason is that this public IP can be changed during start, stop, restart etc. Elastic IP won’t. So, next, lets associate the Elastic IP to this instance.

Associate Elastic IP

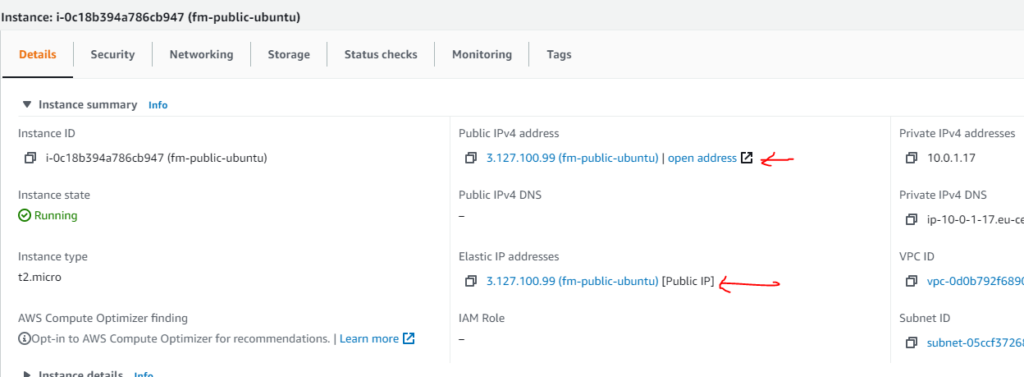
Go to VPC-> Elastic IP dashboard and select the previously created Elastic IP:



Next, you can select the target EC2 instance and complete the association as shown below:



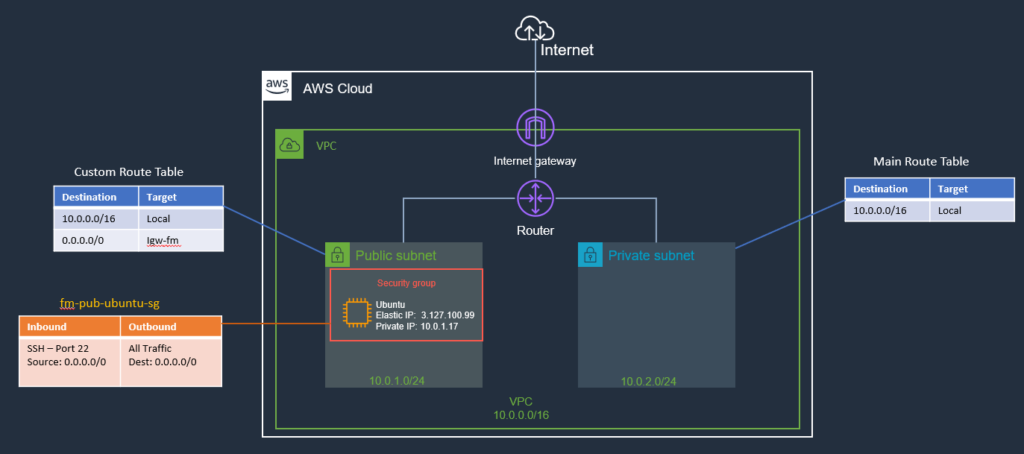
Now, if we go back to **instance detail**, we will see that Elastic IP is populated:



Now, we can use this IP to ssh to our instance as shown before. This is an ubuntu server and you can may be try to do some experiments with it. May be you can try to run an nginx web server with Node.js (you can check my previous post [Use NGINX to Serve .NET Core, Nodejs or Static contents](https://hexquote.com/use-nginx-to-serve-net-core-nodejs-or-static-contents/)). However, I will not be doing anything with it just yet.

Architecture

Let’s have a look on our architecture diagram:

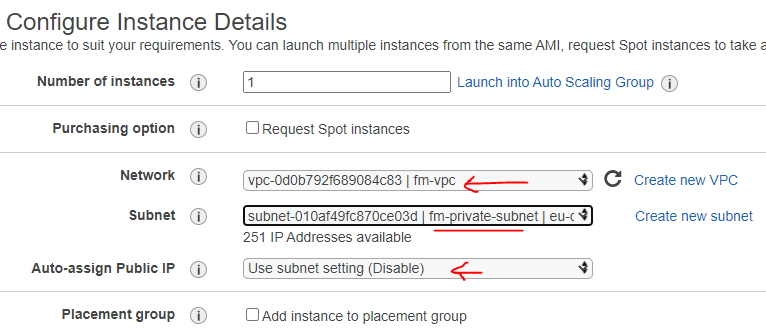


You can see that now this diagram includes the EC2 instance and Security Group Information as well.

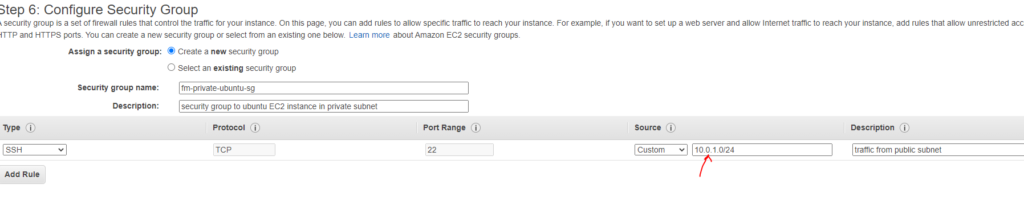
Launch an EC2 instance in Private Subnet

Next, I will create another ubuntu based EC2 instance in the private subnet following the same steps mentioned earlier. However, for this instance, I will not assign any public IP address (Remember, we do not have a route to/from internet in private subnet, so we can not connect to EC2 instance from the internet directly). Typically a bastian-server (aka jump box) is used to connect to instance in private subnet (More on that in later posts).

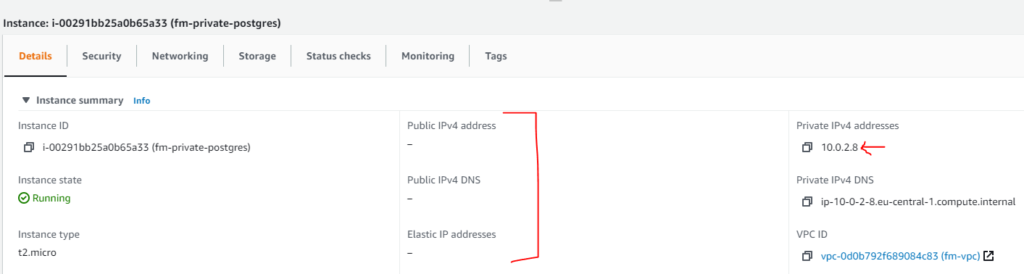
I am thinking of using this EC2 instance in future to host postgreSQL database server. But for now, I will not concern myself with this thought much.



I created a new security group and allow SSH traffic from public subnet (10.0.1.0/24).



Next, Review and Launch the instance and it will be running in few minutes:

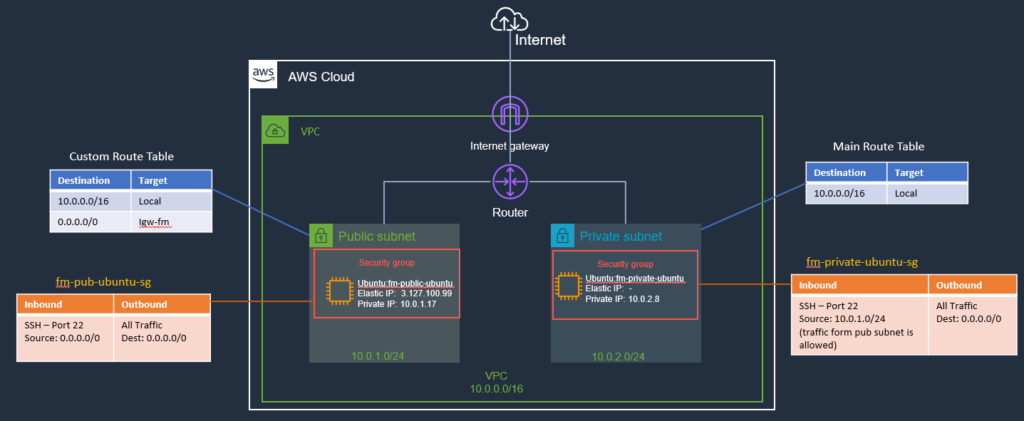


We can see, that this instance has private IP address and no public IP assigned to it.

Now, if we try to SSH to this instance using private IP address, we wouldn’t able to do that because it is in private subnet and there is no direct route from internet to this instance.

But that doesn’t mean that we can not connect to that EC2 instance. What we need to do is to setup a **Bastian Host** (aka Jump Server) to make the connection. We will see an example of how to setup a bastian host in the next post in this series.

Here is how our architecture diagram looks after the changes:



Summary

The core of the web of AWS is EC2. You can think of this as service that **provides servers** where your applications will be running. In AWS they are called EC2 instances instead of servers.

You Launch EC2 instances in a subnet of a VPC. This subnet can be private subnet or public subnet. In this post, we launch two ubuntu EC2 instances; one in public subnet and one in private subnet.

We created one security group for our public EC2 instance, which allows SSH traffic from any source. The second security group which we associate to EC2 instance in private subnet, allows SSH traffic form public subnet only (we will see that it will help us setting up bastian server in the next post). Currently, we can SSH to public EC2 instance using SSH.

Now, you have a linux machine (EC2) in public subnet. You can use it various purposes.

AWS Basics- Bastian Hosts and NATS

March 30, 2021 by [Jawad Hasan Shani](https://hexquote.com/author/jawadhasan80/)

Introduction

In previous post on [AWS Elastic Compute Cloud (EC2) Basics](https://hexquote.com/aws-elastic-compute-cloud-ec2-basics/), we launched two EC2 instances. One in public subnet and one is private subnet. With security groups configured, we were able to SSH to EC2 in public subnet.

In this post, we will continue and setup Bastian Host and NAT instance in our VPC. We will learn why we need those and some of the options available to us.

Here is how our architecture is currently setup, for the reference:

Now, if we want to SSH into EC2 instance on private subnet from our home/office (or using development machine), currently we can’t. Our instance has no public IP, it is in Private Subnet (no direct route from internet). This is where we can use a **Bastian Server**.

Bastian Host

Bastion servers are instances that sit within your public subnet and are typically accessed using SSH or RDP. The purpose of bastian host is to restrict the access to a private network from an external network.

Once remote connectivity has been established with the bastion host, it then acts as a ‘**jump**–**server**’ , allowing you to use SSH or RDP to login to other instances (within private subnets) deeper within your network.

In this post, we will setup a **Bastian server** in our **public subnet** (so internet traffic is possible).

First we will connect to this Bastian server, from there, we can then connect to **private EC2 instance** (remember the rule in security group, we configured, in previous post, we allowed traffic from within our VPC to port 22 on the private EC2 instance).

Setting Up a Bastian Host

It’s agatha all along. We already have a bastian host.

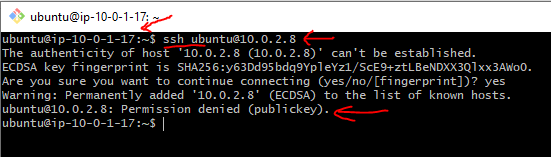
Let me explain this, in the previous post, we launched an **EC2 instance in public subnet** and we already have configured an Internet Gateway for VPC and configured route in route table to allow incoming traffic. Later, we launched an ubuntu **EC2 instance in private subnet** and configured its security group to **allow incoming traffic on port SSH from public subnet** and that is all what was needed on connection level. I just did not mention that in previous post, to not get us distracted from those details. So our current EC2 instance in public subnet can be used as a **Bastian Host**.

The idea is that we SSH to the Bastian instance using its **Elastic IP** (or public IP). Then from the Bastian instance, SSH into private instance using its **Private IP**.

SSH into Bastian Host to Private EC2 Instance

Ok, let’s try this. First we need to SSH into EC2 public instance (we have seen previously, how to do that, please check that post if you need more info.).

Once we are in EC2 machine (bastian-host), from there, we can try to SSH into the EC2 instance on private subnet, run the following SSH command:



Notice, that we still can not SSH into the private instance. The reason for that, is that we also need key-pair file for connection, which is right now missing on the EC2 instance (Bastian Host) on our public subnet.

So, one thing we can do is to copy the keypair file on bastian host using the following command:

scp -i ./fm-keypair.pem fm-keypair.pem ubuntu@3.127.100.99:/home/mykeys

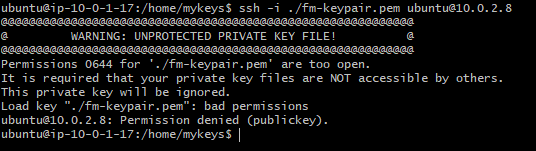
we can check that keypair file is now copied to EC2:



Now, if we try to SSH using this key file:

ssh -i ./fm-keypair.pem ubuntu@10.0.2.8

This may result in the following error message, ok it is trying, but still can’t SSH.

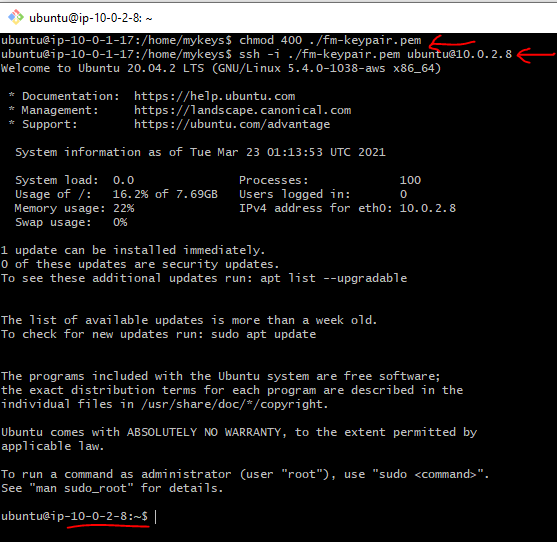


To fix this, we need to set permission on the key-pair file using **chmod** as shown below:

chmod 400 ./fm-keypair.pem

This command will **change the permission of that file** so it is only readable by us.

Ok, let’s try again to SSH to private ec2 instance and this time we are in (notice the private IP changes in the prompt):



Ok, so, Bastian Host, allowed us to connect to EC2 instance in private subnet. We can now use it to jump to EC2 instances in the private subnet.

Outbound Internet Access from Private EC2 instance

Ok, we are able to connect to private EC2 from internet via Bastian host. Incoming traffic is working as expected.

Now, I would like to install some software e.g. PostgreSQL on this private EC2 instance from internet (your may not have this requirement), meaning I want to access the internet from this instance. Can we do that? Let’s try this next:

I tried to **curl *google.com*** from private EC2 instance(you can try may be **apt-get update** command instead):



and as you can see that it just hangs there and nothing happens.

This is expected and is a good thing. If you remembered from previous posts, the whole point of private subnet is that it shall not be connected to internet directly (Inbound/Outbound) and this is what we are experiencing here.

Now, without internet, we can’t do much on this instance. For example, I can not download the package from internet. So we need a way to allow outbound internet traffic from this private EC2 instance.

Bastian Host job was to provide us a mean to **SSH into** private instance and we are able to do that. Now, if we want to **go out to** internet from private EC2 instance, we can use a **NAT instance** or **NAT Gateway** to give access to the internet without allowing inbound access to it.

What is a NAT Instance

A NAT (Network Address Translation) instance is, like a bastion host, an instance that lives in your public subnet.

A NAT instance, however, allows your private instances outgoing connectivity to the Internet, while at the same time blocking inbound traffic from the Internet.

Main reason to configure NAT instances is to allow private instances to access the Internet for important operating system updates, It is used for purposes like patching your OS etc.

Setting Up a NAT Instance

It is actually very simple to setup a NAT instance.

First, we will launch an EC2 instance in our public subnet. We also need to allocate an Elastic IP address and assign it to NAT instance as well. This will allow that our instance is be able to reach the internet. We will be using a special amazon image instead of ubuntu for our NAT instance.

Second, in order for our private EC2 instance to use the NAT instance to get to the internet, we need to configure a route in the Route table associated with our private subnet, pointing to the NAT instance as target.

Third, we need to disable IP source/destination check on the NAT instance.

So Traffic flow will be from: private EC2 –> Route Table –> NAT instance –>Internet Gateway –> Internet.

Question:

**Can we use our Bastian-host as a NAT instance as well?**

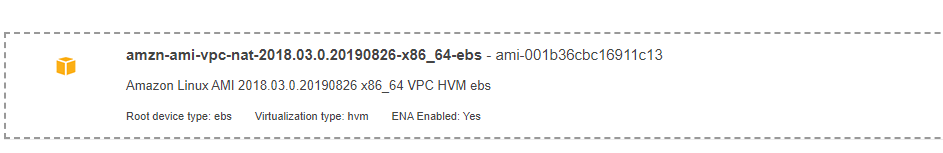
Answer:

Yes, that is totally fine. However, NAT instance also require some IP table configurations to allow this behavior and currently I am using ubuntu for Bastian Server and it will require me to run some bash commands on it to setup the IP table entries and I really don’t wanna do this at this time.

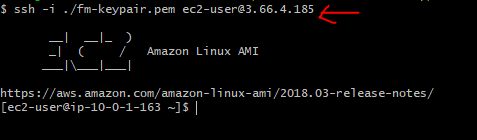
Another easy option is that there are some pre-configured amazon images you can use as a base for NAT instance and I am going to use this option for NAT instance (Later may be I will use this instance as a Bastian host and remove the ubuntu EC2 from the public subnet all together). For now, let’s keep things simple and use this pre-configured Amazon AMI based EC2 instance.

Launch the EC2 instance (NAT instance)

So, launch the EC2 instance wizard from the web console. Use the following image for NAT instance (you can use **ami id** in the search field to search for this instance):



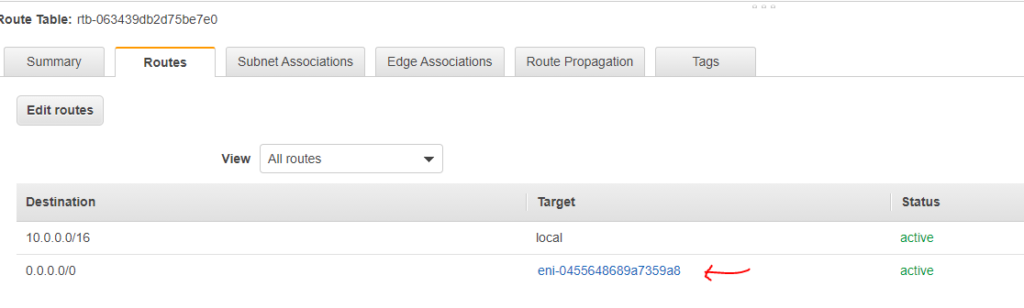
select the right **VPC** and choose **Public Subnet** for our **NAT**instance (like our bastian-host) and launch the instance. If you like you can try to SSH into this instance:



So our instance is running and all the needed IP table stuff is already done for us.

Configure the Route for Private Subnet Route Table

Next, go to the route table for the Private subnet and**add one more route (0.0.0.0/0)** and for **target**select the **NAT Instance eni** we just created. This way we are configuring a catch-all route in our private subnet and sending traffic to NAT instance which is in public subnet:

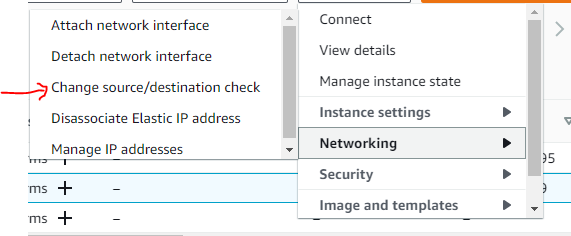


So, any time private EC2 instance make an internet request, it will go to ENI of NAT instance and NAT instance being in public subnet and have a route configured to internet gateway will be able to reach out on the internet.

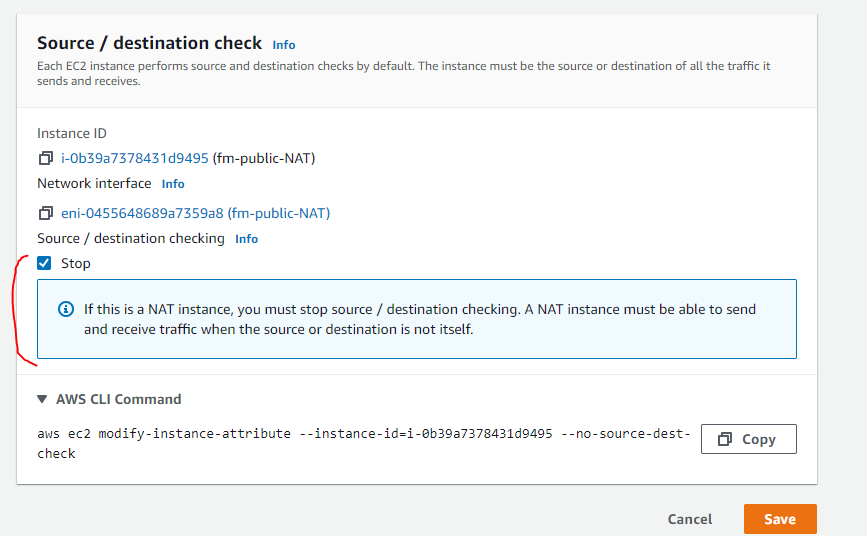
Now, if we launch more EC2 instances in the private subnet, they can all use this mechanism to reach out to internet. So NAT instance will be providing a single public IP for all these private IPs.

Disable Source/Destination Checks for NAT instance

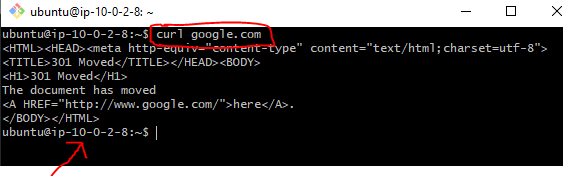
Select the NAT instance we created above from the EC2 web-console dashboard and then from the Menu bar, choose the option to **Change source/destination check** as shown below:



Select the Stop checkbox if it is not selected and there is also some info about why this is needed.



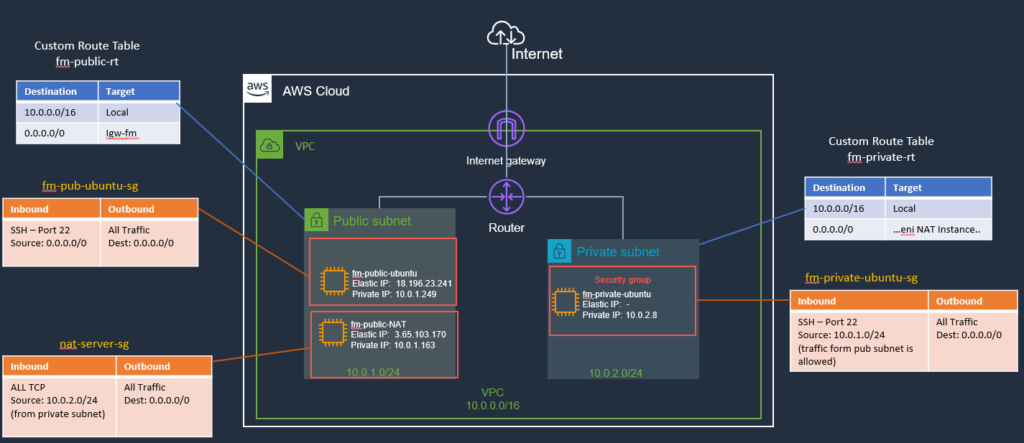
Now, lets try again the curl command from private EC2 Instance and this time, it works:



Cool, our NAT instance is also setup now. Let’s look at the updated architecture diagram next.

Architecture Diagram

I also updated some of the names, Security groups, Route Table and Inbound/Outbound rules. I will cover Security Groups in details in later post. Here is how our architecture currently looks like:



Summary

Bastion host and NAT instance both help secure your AWS infrastructure by disallowing/limiting access to your instances over Cloud.