Cloud Infrastructure HS 21

AWS

by Dejan Jovicic and Thomas Kleb

Contents

1 Introduction

For this lab we were given a $100 budget to work with the Amazon Web Services (AWS) to setup and maintain a simple website. The lab was structured with different steps of setting up and getting to know the Interfaces provided by Amazon. We created a static website to fiddle around with the settings and later upgraded it to a dynamic one using Amazon Elastic Compute Cloud (Amazon EC2) with the correct access parameters to connect. To create a certain redundancy and security for the businesses information we then learned how to migrate the database which runs on the EC2 to a database running on the Amazon Relational Database Service (RDS). At last additional availability and scalability insurances were added and tested to work. This lab gave us a first insight into how AWS works, and which settings are important to setup a simple webservice running a website. This report only consists of the screenshots we deemed as important for the steps (mostly consisting of finicky settings). We screenshotted every step of the processes and added them in a separate file on our GIT to follow our steps exactly.

2 Creating a Static Website for the Café

The goal of this lab was to have a static website using Amazon S3 and implementing a data lifecycle strategy as well as a disaster recovery strategy. At the end we could access the architecture via this link:

http://klebjov-cafe.s3-website-us-east-1.amazonaws.com/index.html

2.1 Setting up an Amazon S3 Bucket

The first step of creating a static website is to create the bucket in which the files will be placed to run. This bucket is needed to upload our data to Amazon S3. This could be performed either by using the console or the GUI which both are provided by Amazon to manage our uploads. We chose to use the GUI and proceeded to create a bucket by selecting the option on the site. The first step was to give a globally unique name to our bucket (“klebjov-cafe”) and decide which region we host it from (“us-east-1”). Since it wasn’t important, we decided to allow all public access to our website. It is recommended to turn on all four settings to block public access for all current and future buckets (application has to work without public access, which wasn’t the case for this lab).

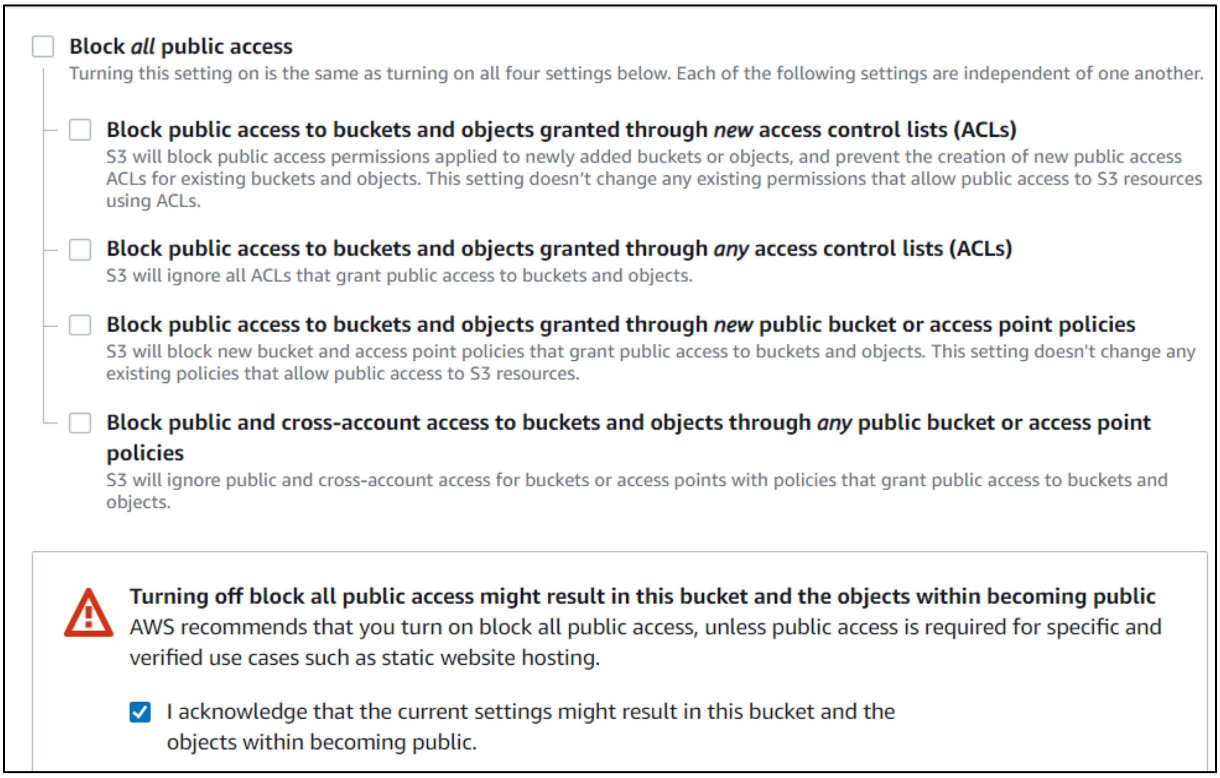


Figure 1 Managing public access to bucket

As additional settings we enabled versioning and disabled encryption to make it easier to handle files on our webservice.

Versioning, when enabled, allows all the objects added to the bucket to receive a unique version ID which is “null” when versioning is disabled. This allows keeping of multiple variants of an object in the same bucket. Furthermore, this feature makes it easy to preserve, retrieve and restore every version of every object stored in our bucket (easier recovery if an unintended action or an application failure takes place).

For this lab encryption wasn’t needed so we disabled the default encryption to prevent any potential mishaps. The default encryption of AWS buckets uses server-side encryption with either Amazon S3-managed keys (SSE-S3) or the AWS Key Management Service keys (AWS KMS). If we had to change the default encryption feature in a future update / step of the lab, additional request charges would be needed which we wanted to avoid.

2.2 Adding Index, CSS and Images

The next step was to start the static website hosting by adding the given index.html to the settings. After that we had our interface to upload additional files to our webservice.

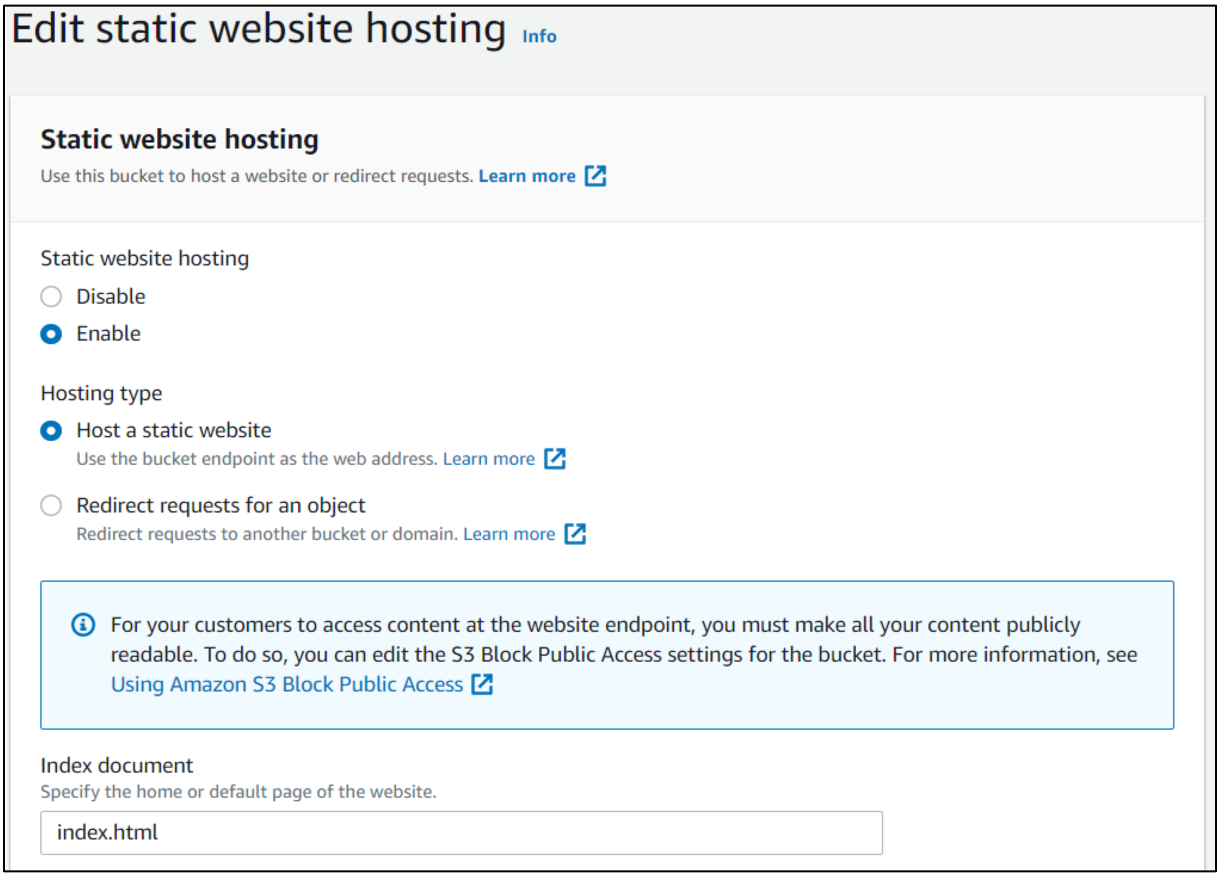


Figure 2 Static Website Hosting Settings

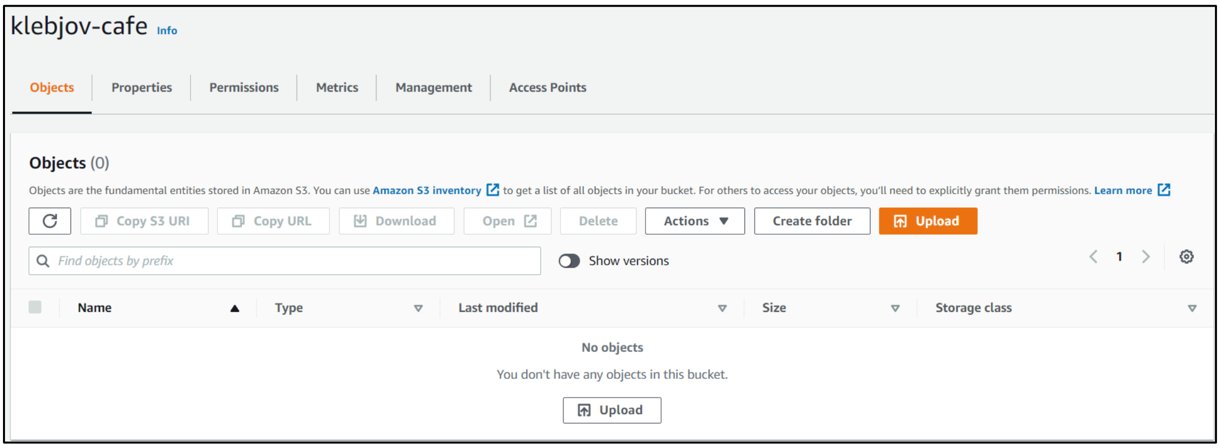


Figure 3 Interface of the Webservice

Then we granted public read access to the service:

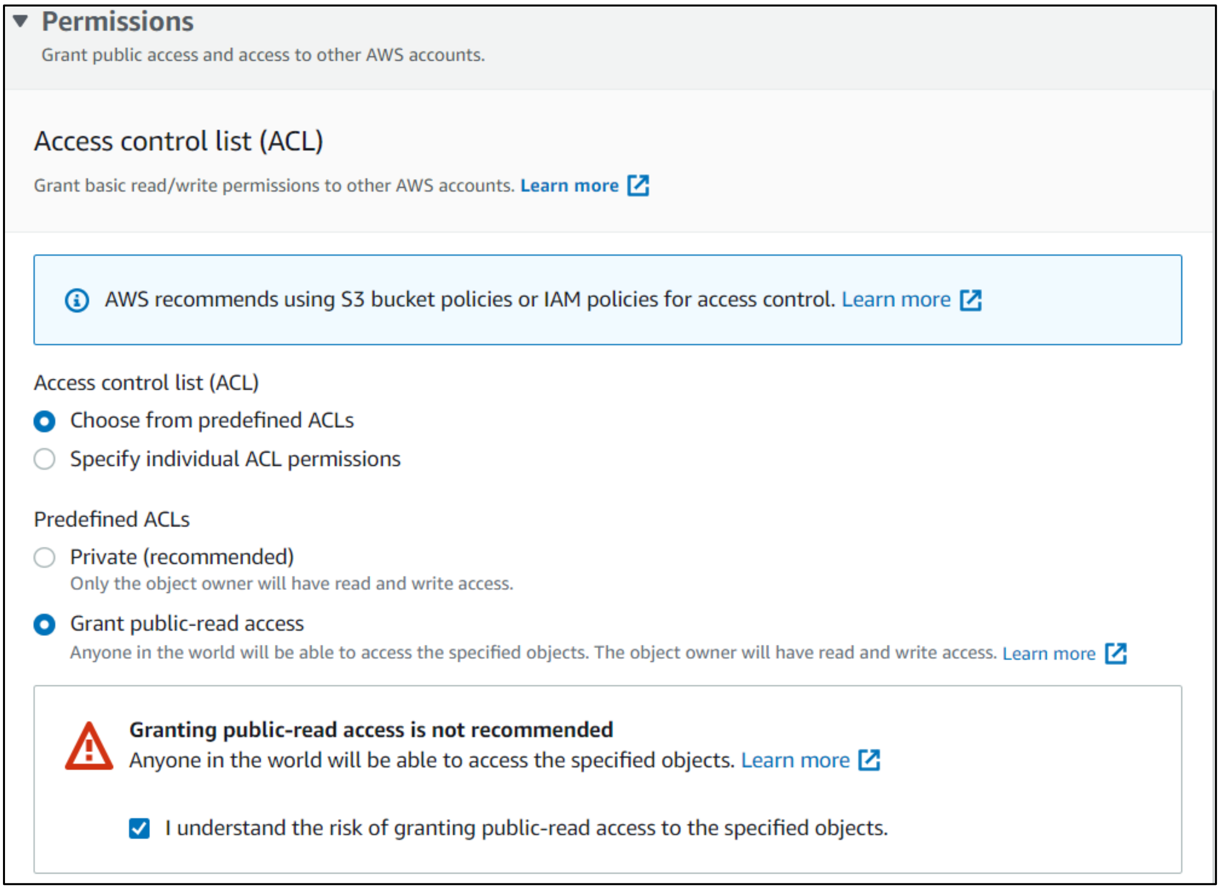


Figure 4 Adding public-read access Acknowledgement

2.3 Updating Bucket Policy

Next, we created a bucket policy which automates the process of giving an uploaded object the public access permission setting, so we don't have to do this manually. To achieve that we created a bucket policy which is in JSON format. To get to the tab we clicked on "Permissions". We used the policy generator and edited the file to make it fit our goals.

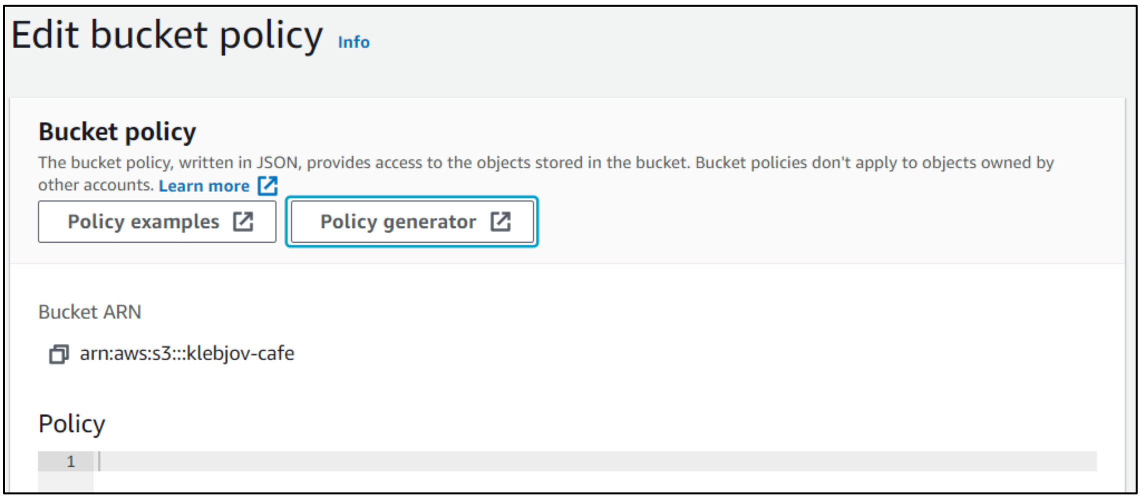


Figure 5 Bucket Policy Settings

The final policy looked as followed:

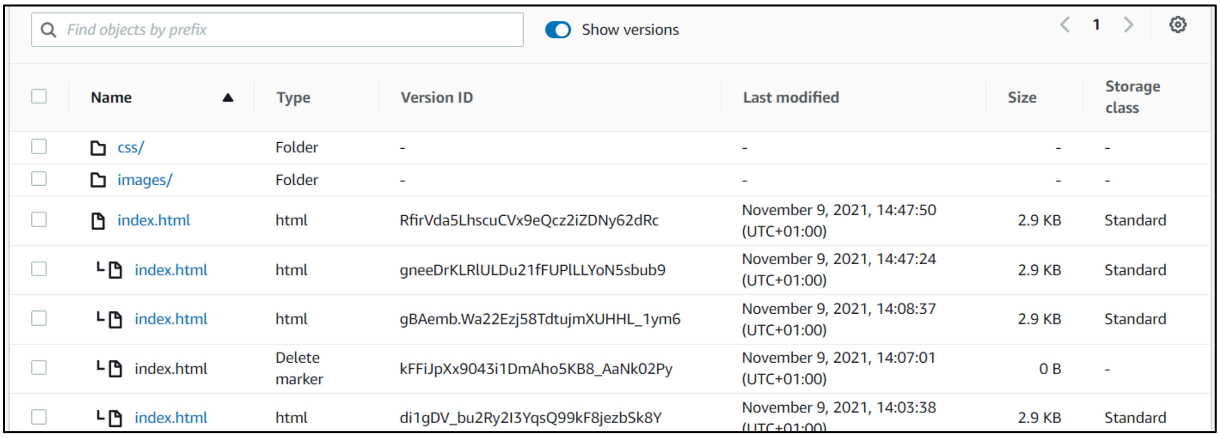
```
{
    "Version": "2012-10-17",
    "Id": "Policy1636464504009",
    "Statement": [
        {
            "Sid": "Stmt1636464498108",
            "Effect": "Allow",
            "Principal": "*",
            "Action": "s3:GetObject",
            "Resource": "arn:aws:s3:::klebjov-cafe/images/*"
        }
    ]
}
```

2.4 Prevent accidental overwriting

To prevent accidental deletion or overwriting of files we use a feature we already enabled before: Versioning. This doesn't allow the direct deletion of an object. All versions remain in the bucket and a delete marker is introduced which becomes the current version. If we needed to delete an object, we need to remove that delete marker too. Existing objects in the bucket do not change and only future requests behaviour changes. If we put an object retrieval request, the current version of the object will always return. This allows for easy rollbacks.

If a file is overwritten, the bucket keeps the old version but uses the new one as the current. Rollback is as easy as with deletion.

The updates of each file can be seen by enabling the "show versions" button:



Name	Type	Version ID	Last modified	Size	Storage class
css/	Folder	-	-	-	-
images/	Folder	-	-	-	-
index.html	html	RfirVda5LhscuCVx9eQcz2iZDNy62dRc	November 9, 2021, 14:47:50 (UTC+01:00)	2.9 KB	Standard
index.html	html	gneeDrKLRlULDu21fFUPlLLYoN5sbub9	November 9, 2021, 14:47:24 (UTC+01:00)	2.9 KB	Standard
index.html	html	gBAemb.Wa22Ezj58TdtujmXUHHL_1ym6	November 9, 2021, 14:08:37 (UTC+01:00)	2.9 KB	Standard
index.html	Delete marker	kFFiJpXx9043i1DmAho5KB8_AaNk02Py	November 9, 2021, 14:07:01 (UTC+01:00)	0 B	-
index.html	html	di1gDV_bu2Ry2I3YqsQ99kF8jezbSk8Y	November 9, 2021, 14:03:38 (UTC+01:00)	2.9 KB	Standard

Figure 6 Enabling Versioning

2.5 Saving costs by adding lifecycles

The last step is to add lifecycle rules to our bucket which ensure that older versions retire after a predefined time. After 30 days the policy moves older versions of the objects in our source bucket to a cheaper storage tier and expires them after 365 days. These transitions have to be configured separately.

This means for both the 30- and the 365-day policy we have to create a lifecycle rule which is applied to all the objects in the bucket.

The 30-day rule has following configurations:

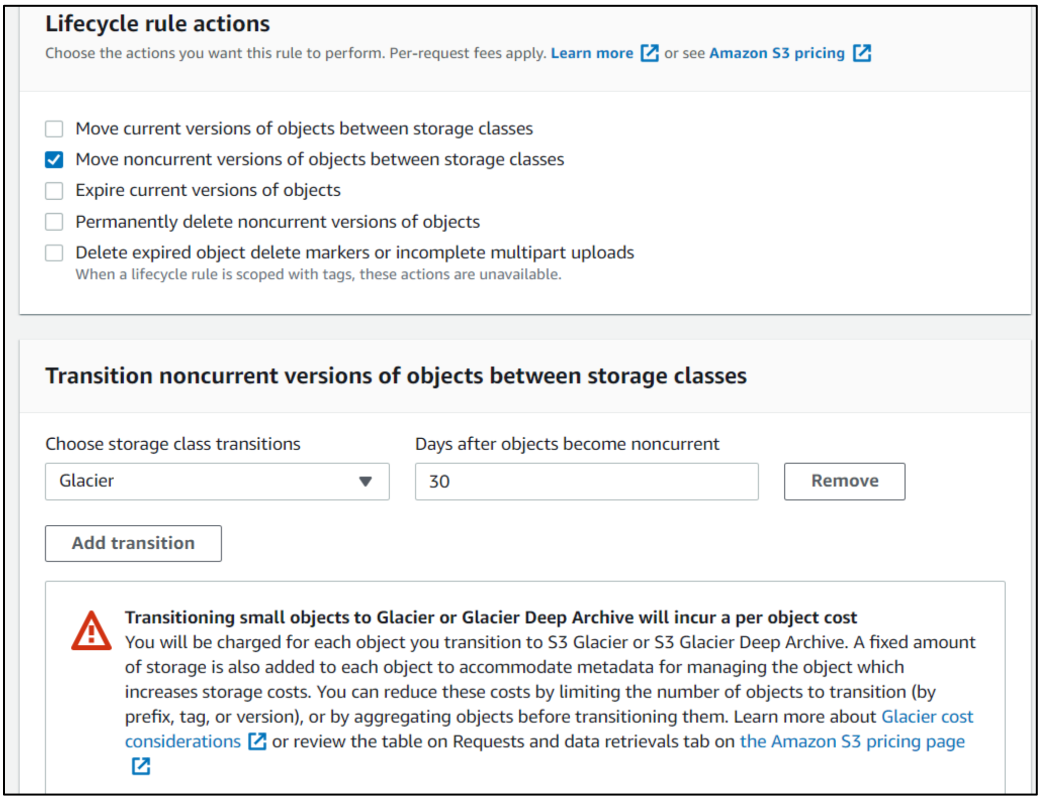
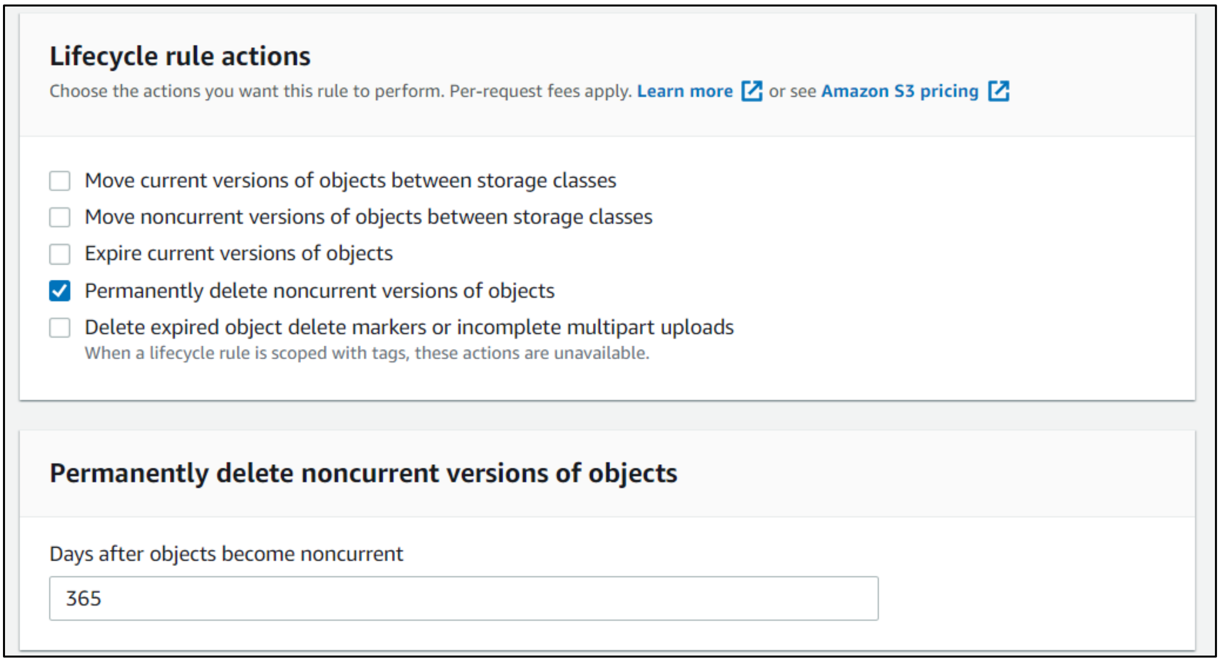


Figure 7 30-day lifecycle rule configs

The 365-day rule has following configurations:



3 Creating a Dynamic Website with the CldInf App

In this part of the lab, we deployed an application on an Amazon Elastic Compute Cloud (EC2) instance. The application shows us all the past events while storing our website access as a timestamp in the database. The goal was to setup said EC2 instance and enable the access to the internet in AWS. Additionally, we had to install the application which was given in a folder.

The final architecture of our system looked like this:

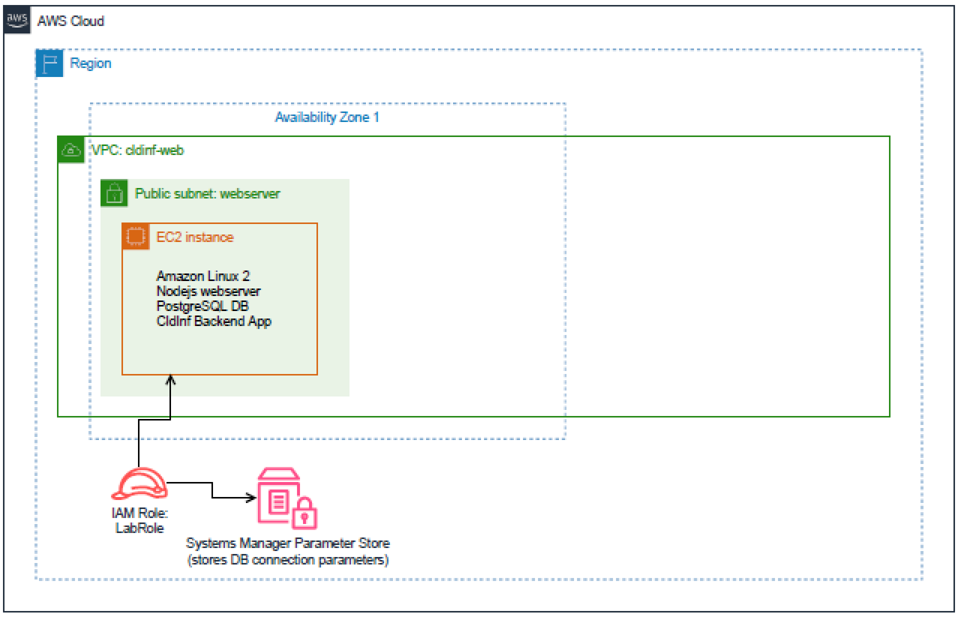


Figure 8 Dynamic Website Example

3.1 Setting up the Instance

After a new instance was created, we configured the Amazon Linux 2 settings. First, we selected the SSD Volume Type to be 64-bit (x86) and selected the t2 micro option.

T2 instances are a low-cost, general-purpose type which provide a baseline CPU performance but can be further improved if we needed to. These are ideal for a general-purpose application like the one we are installing (micro-services, virtual machines, low-latency interactive applications to name a few). The one we selected, t2.micro, has 1 vCPU with 1 GiB of RAM

To name the EC2 we created a tag and gave it the value “CldInfServer”.

Next, we setup the instance details which need a VPC and a Subnet to be configured correctly. At this moment these were on the default settings which isn’t correct.

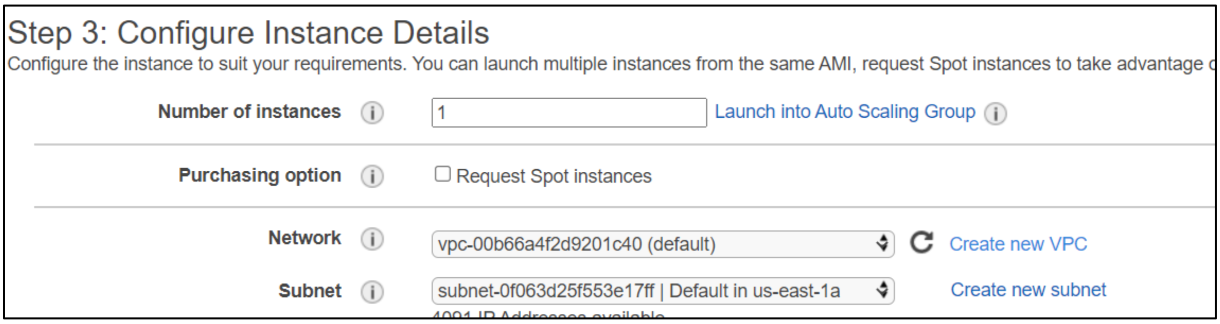


Figure 9 EC2 Instance Details configuration

3.2 Creating VPC

The Virtual Private Cloud (VPC) gives us control over the virtual networking environment with additional features like connectivity control and security management. As seen in figure 8, the EC2 and RDS instance is added to the VPC They can also be configured to control different availability zones and allow communication / connectivity between them.

The VPC is called “cldinf-web” and we gave it the IPv4 CIDR block 172.32.0.0/16 with the other settings staying default.

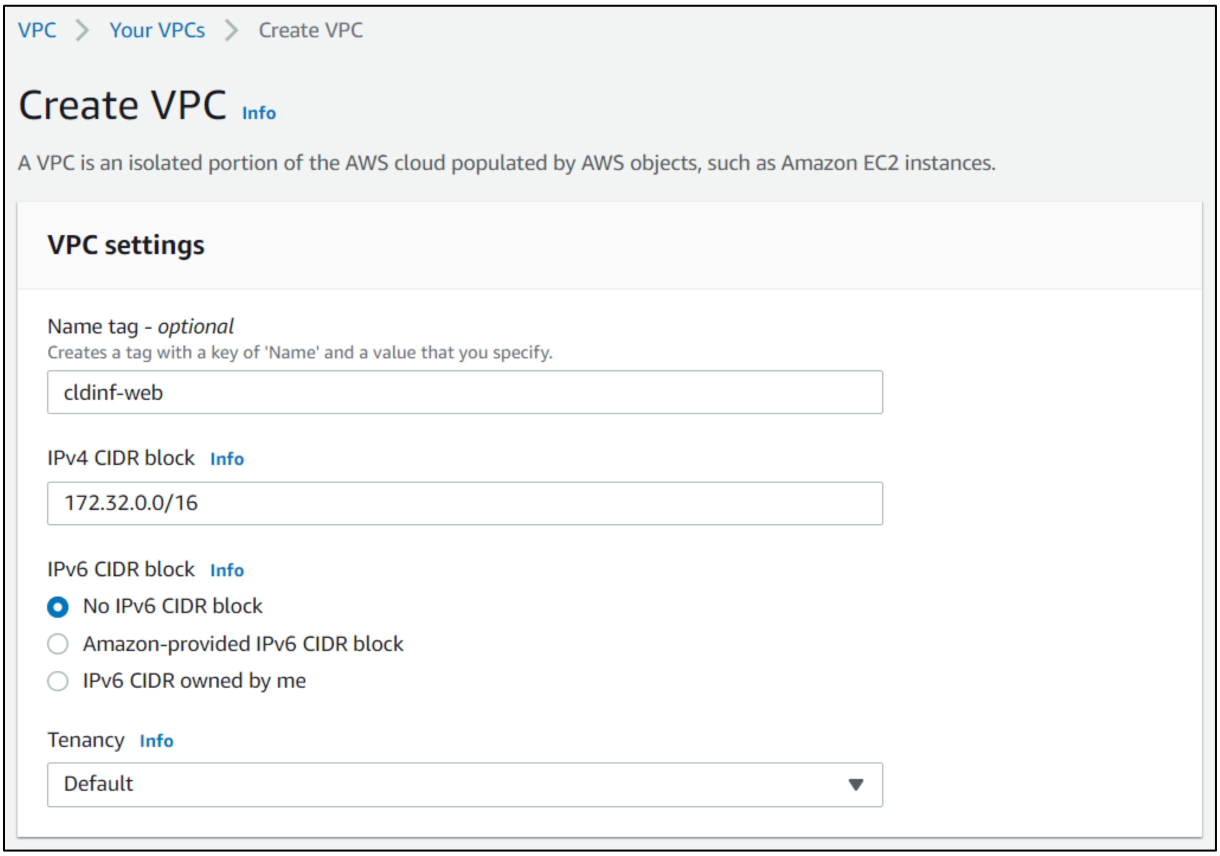


Figure 10 Creating VPC "cldinf-web"

3.3 Creating Subnet

First, we chose the VPC in which we create the subnet. And configured it as follows:

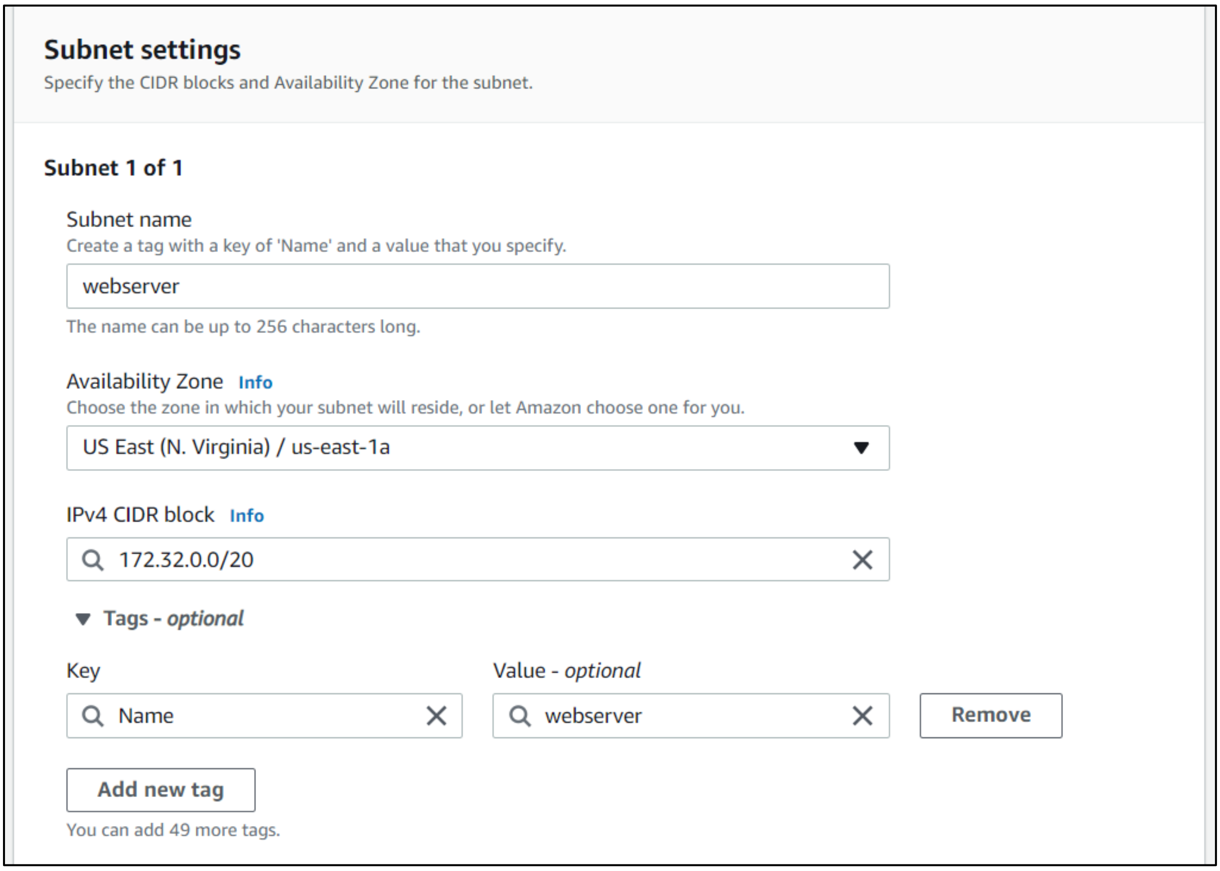


Figure 11 Subnet "webserver" configurations

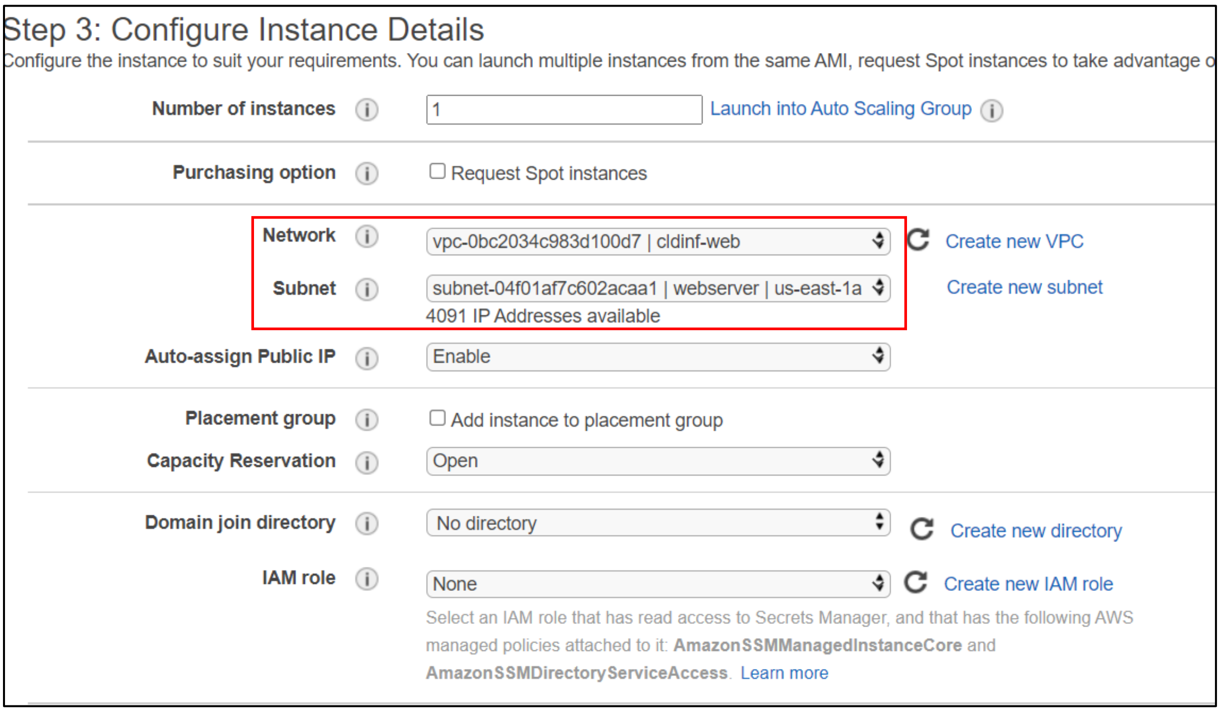


Figure 12 Finished EC2 Details configuration

As key pair we could use the already provided “vockey” and launch the instance.

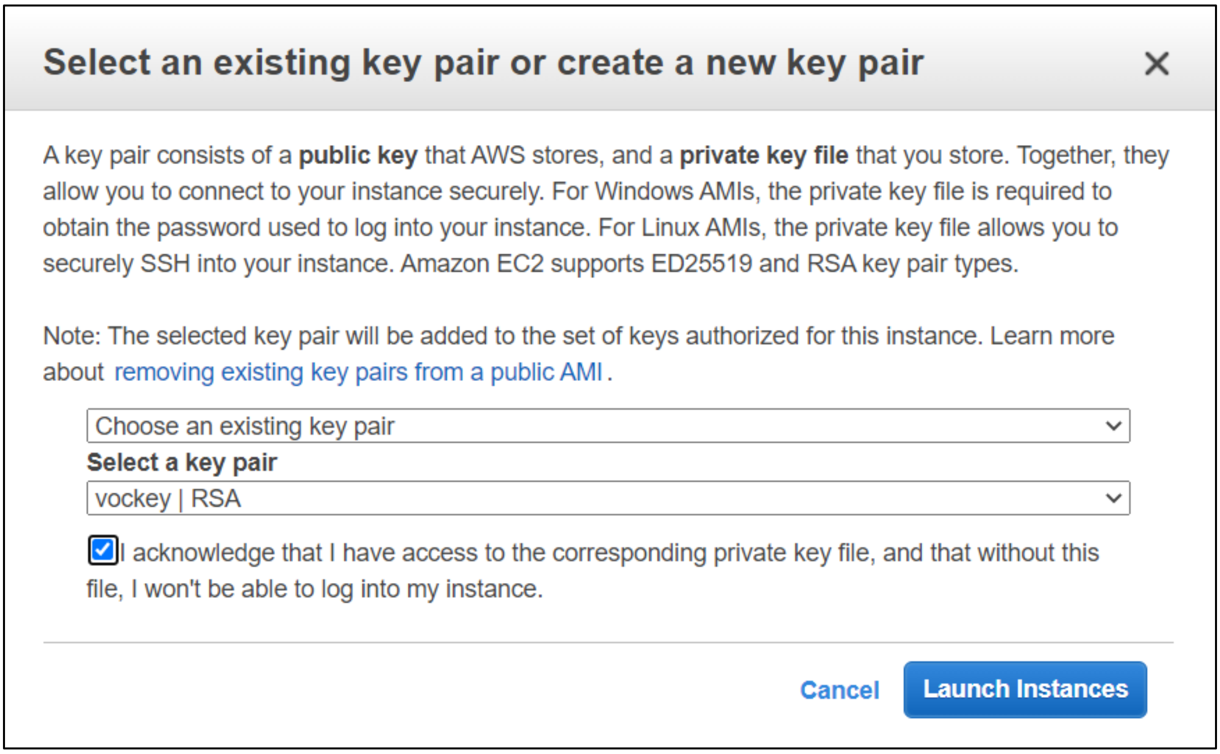


Figure 13 Adding "vockey" key

3.4 Enable Connectivity with Internet

3.4.1 Adding Elastic IP Address

To allow the instance to be accessed from the internet we have to assign a public IPv4 address to it. Amazon allows the use of Elastic addresses which are reachable from the internet and connect to it with our own computer. The practical thing about these addresses is, that these are automatically associated out of a pool, so we don’t have to think about possible intersection.

Like everything else in this lab, the IP address are assigned to the “us-east-1” network border group

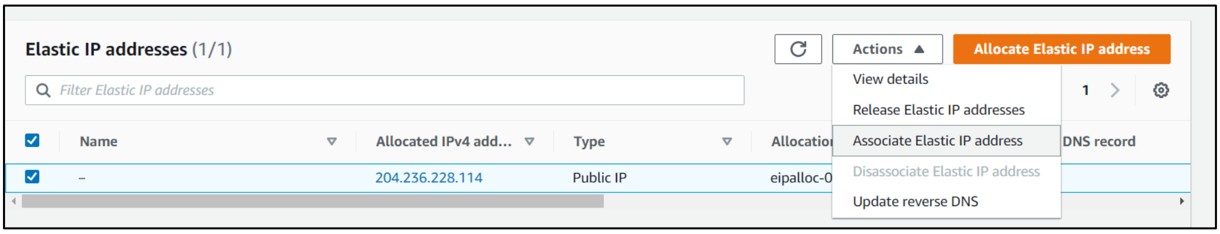


Figure 14 Elastic IP address Interface to control them with different Actions

After generating the public IP address, we can associate it with a private one on our network to allow access from the internet.

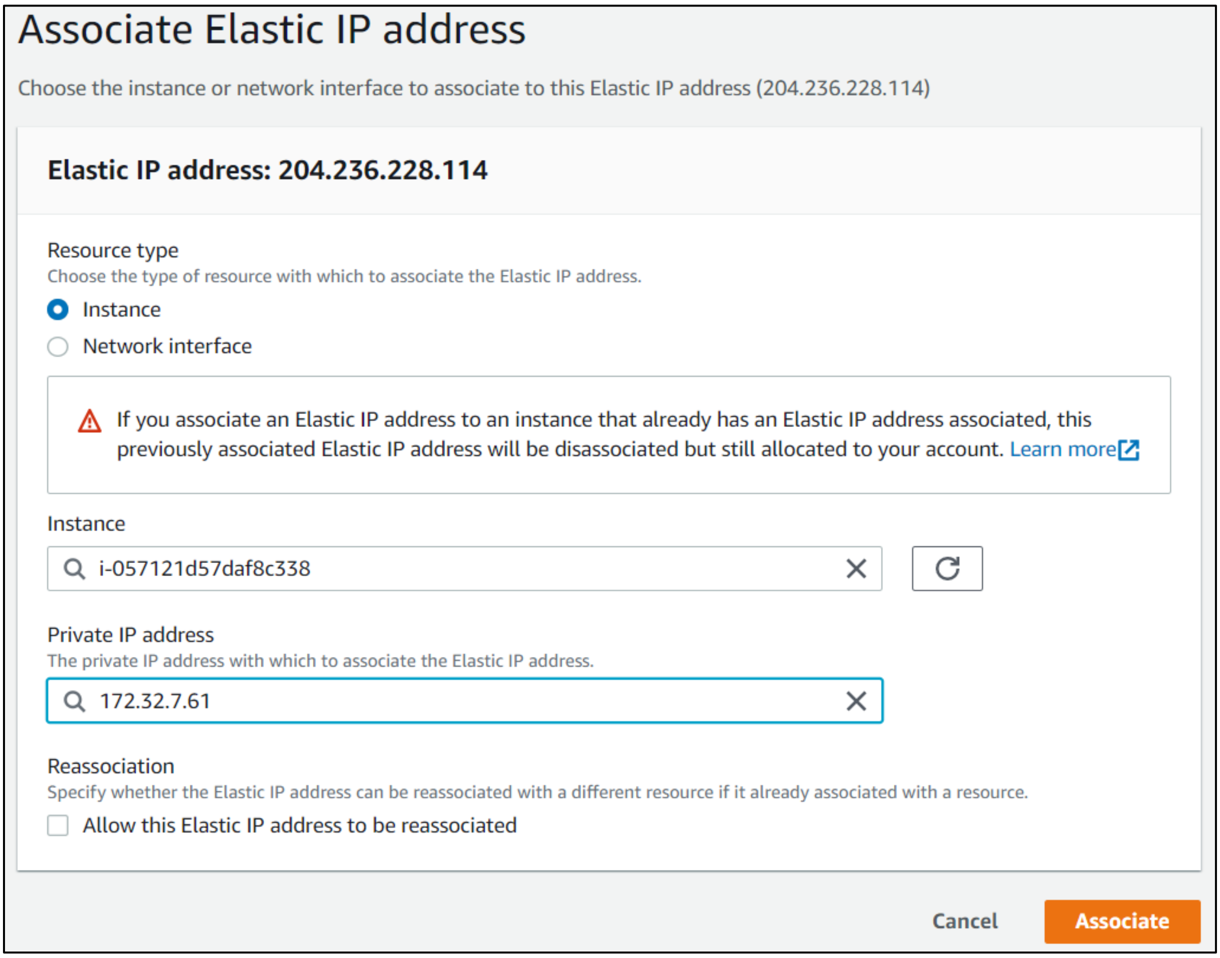


Figure 15 Associating public IP to private IP

3.4.2 Creating Internet Gateway

Next, we had to enable an internet gateway for our VPC. There was already one configured but for another VPC. We had to create a new one.

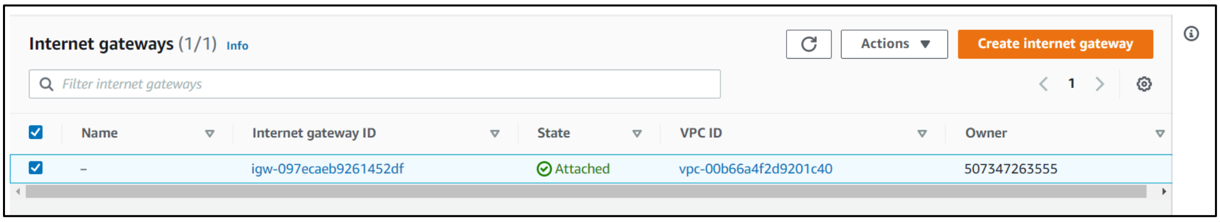


Figure 16 Default Gateway Control Panel

The configuration of an internet gateway is thanks to the simple options of the AWS interface rather simple. We only had to name it, add tags, and attach it to our VPC.

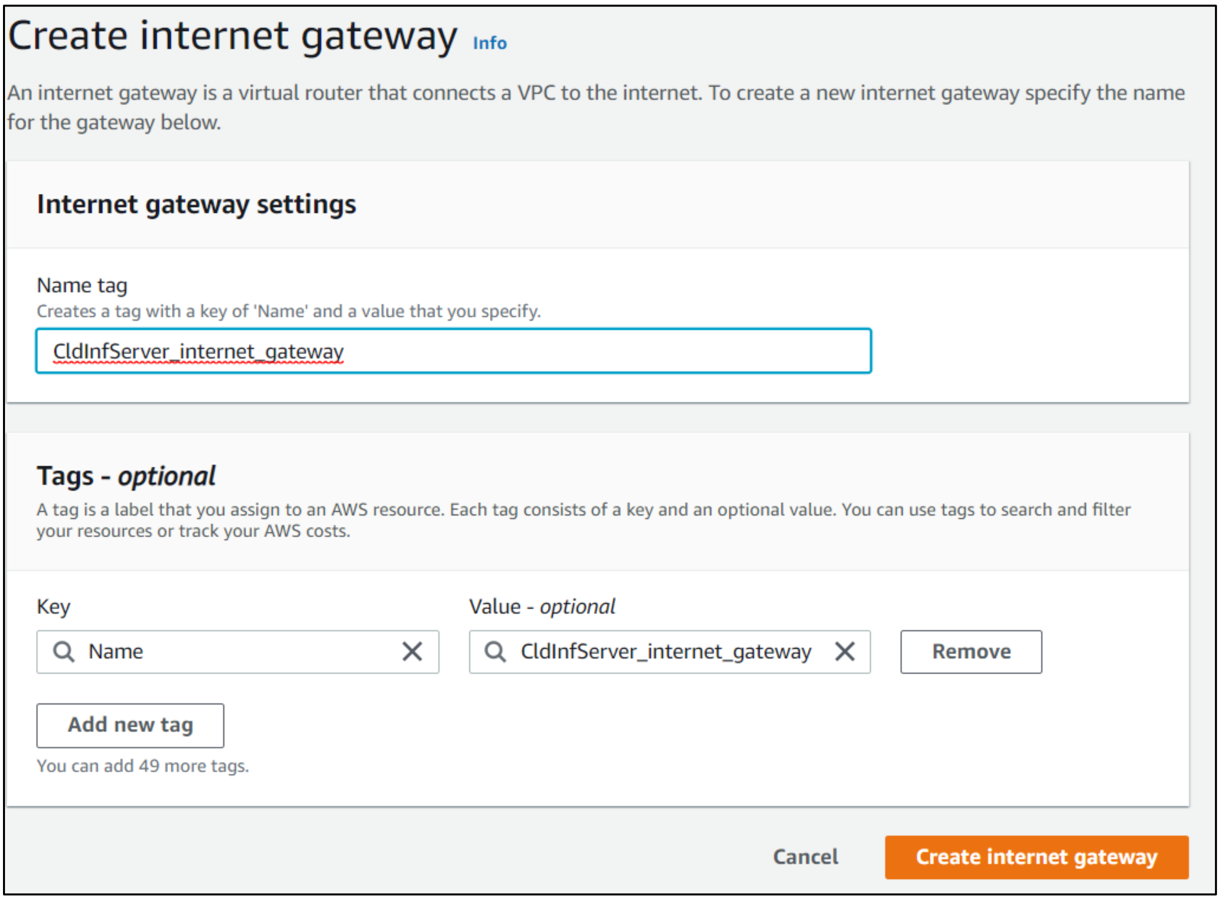


Figure 17 Creating Internet Gateway

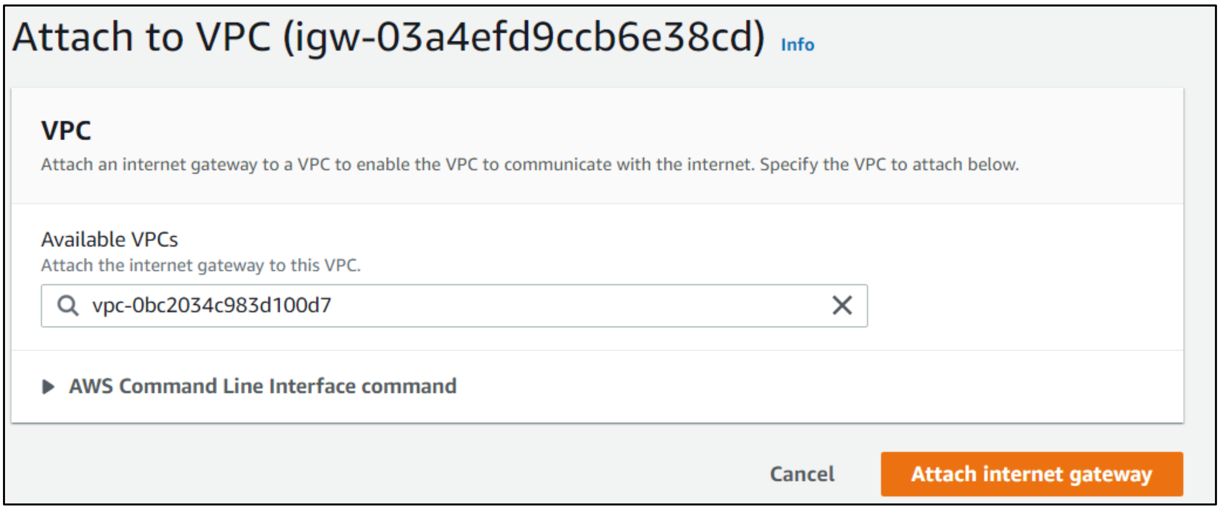


Figure 18 Attaching Gateway to VPC

3.4.3 Adding Security Group

Here we created a security group and allowed SSH (TCP port 22) and HTTP connections (TCP port 80) from the Public IP range of OST (152.96.0.0/16) into the inbound rule-set.

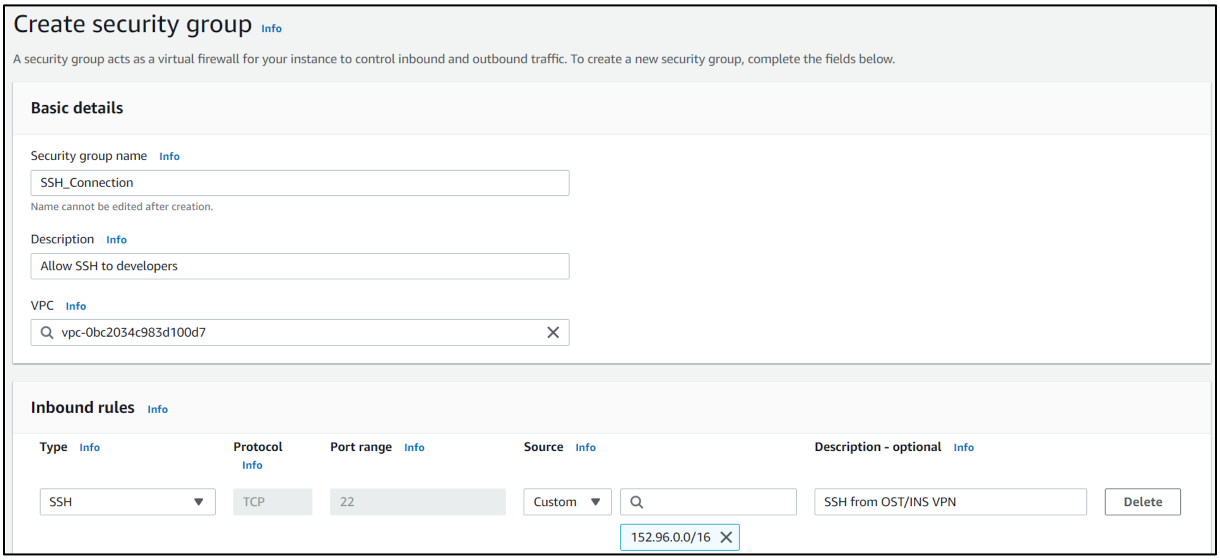


Figure 19 Adding Security Group for SSH Connection

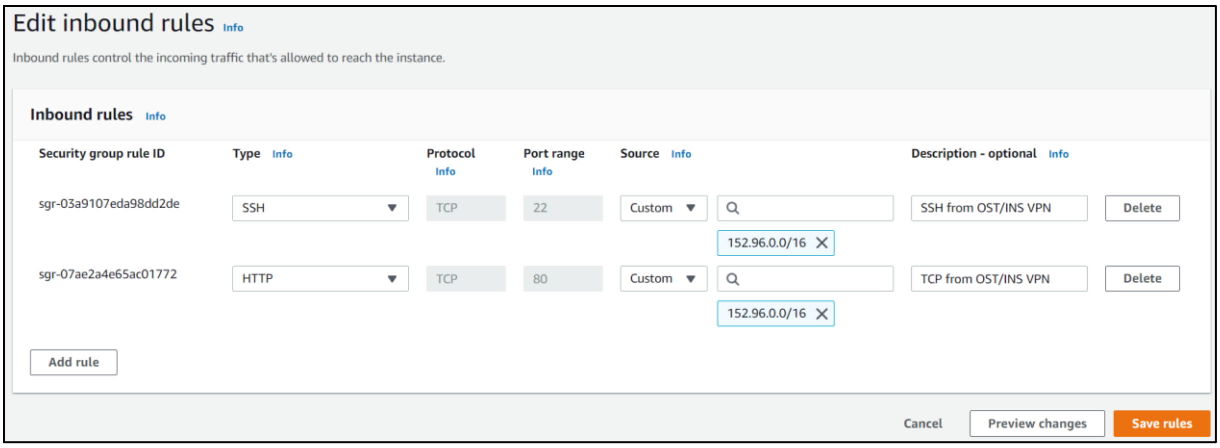


Figure 20 Adding Security Group for HTTP Connection

3.5 Setup SSH on the Client Side with PUTTY

Amazon uses public key cryptography to encrypt and decrypt login information. This means a public key is used to encrypt data while the recipient can use the private key to decrypt said data. For our problem this means that AWS stores a public key, and we download the Privacy Enhanced Mail (PEM) file which is a widely used X.509 encoding format for security certificates. These two together allow us to securely connect to our AWS via SSH.

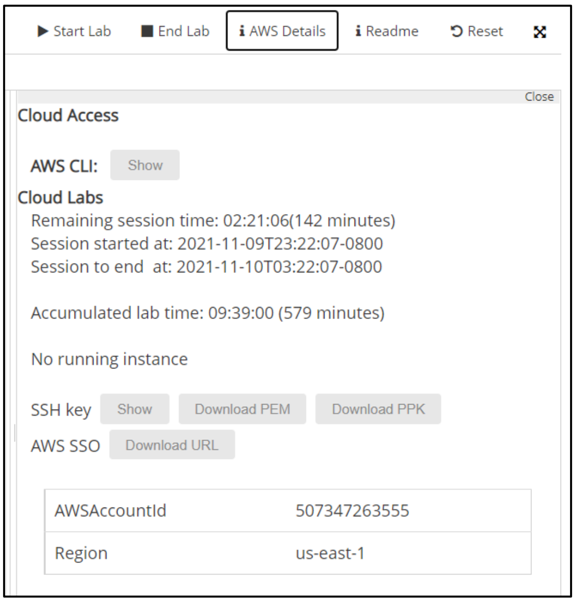


Figure 21 Download of the AWS file

The PEM file then has to be uploaded into puttygen to save it as a putty private key (.ppk). And then loaded into the authentication folder in putty ("Connection/SSH/Auth")

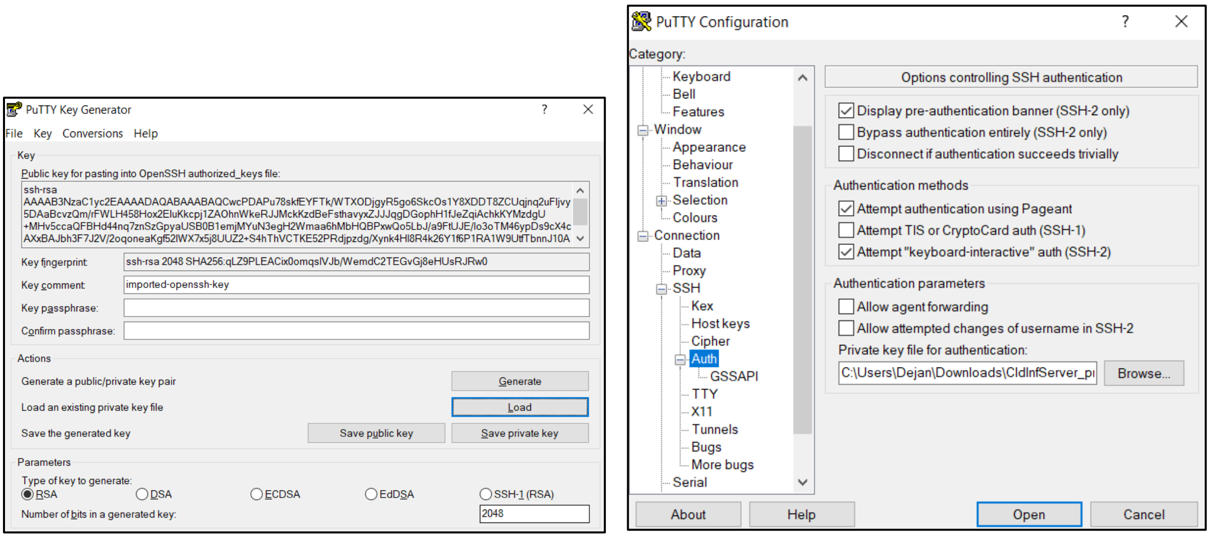


Figure 22 Generation of .ppk file and uploading to putty authentication folder

Then we had to set the ec2-user (standard user) in the “Connection/Data” folder of putty and under the session tab connect via SSH to the public IP from (figure 14 on page 11) the service with port 22) 11

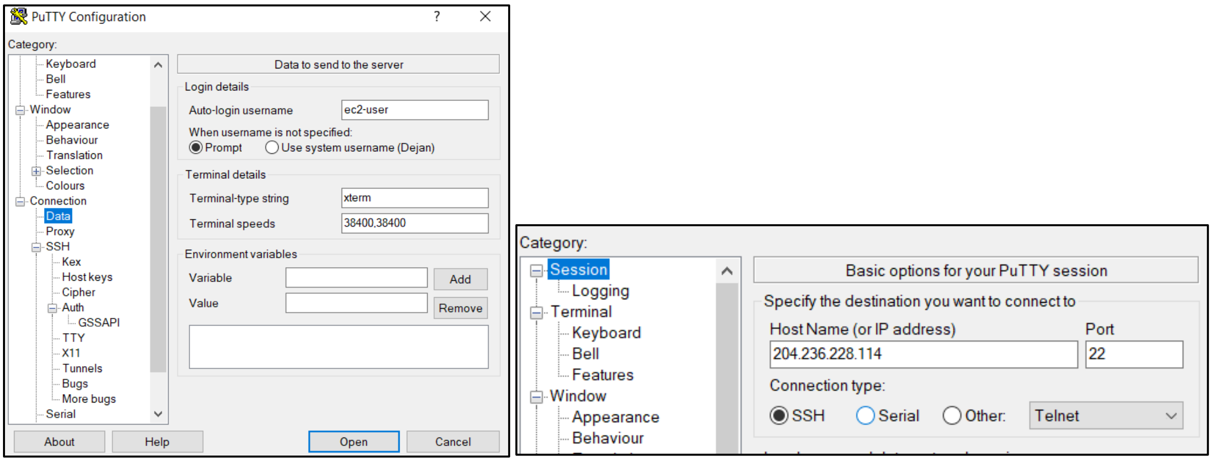


Figure 23 Adding user and connecting to service

3.6 Setup DNS Hostnames

We have enabled a DNS hostname for our EC2 Instance by going to “Your VPCs -> Actions -> Edit DNS hostnames”. With this we can access our website through the DNS hostname, instead of the Public IP

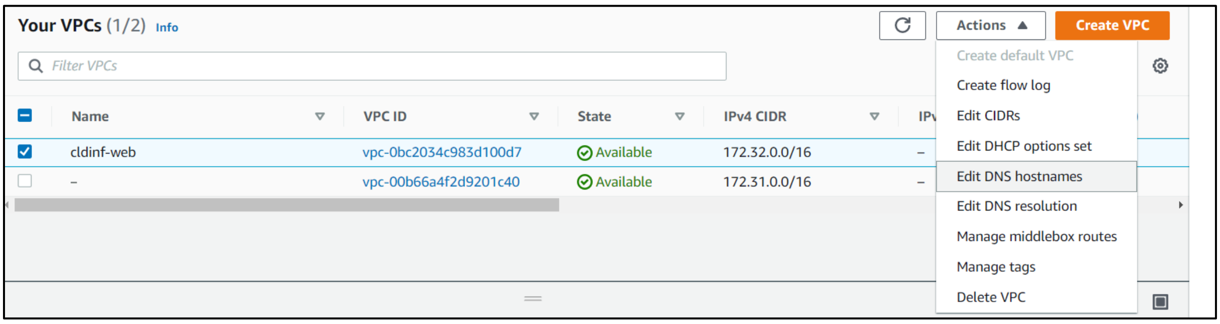


Figure 24 Setting up DNS hostnames

3.7 Create and Associate Route Tables

The route can be created by naming it and attaching it to our “cldinf-web” VPC. Then we edited the route table to have the route 0.0.0.0/0 (anywhere) target the internet gateway we have created:

Figure 25 Route Table configuration

Associate the subnet which we have created before with the route table

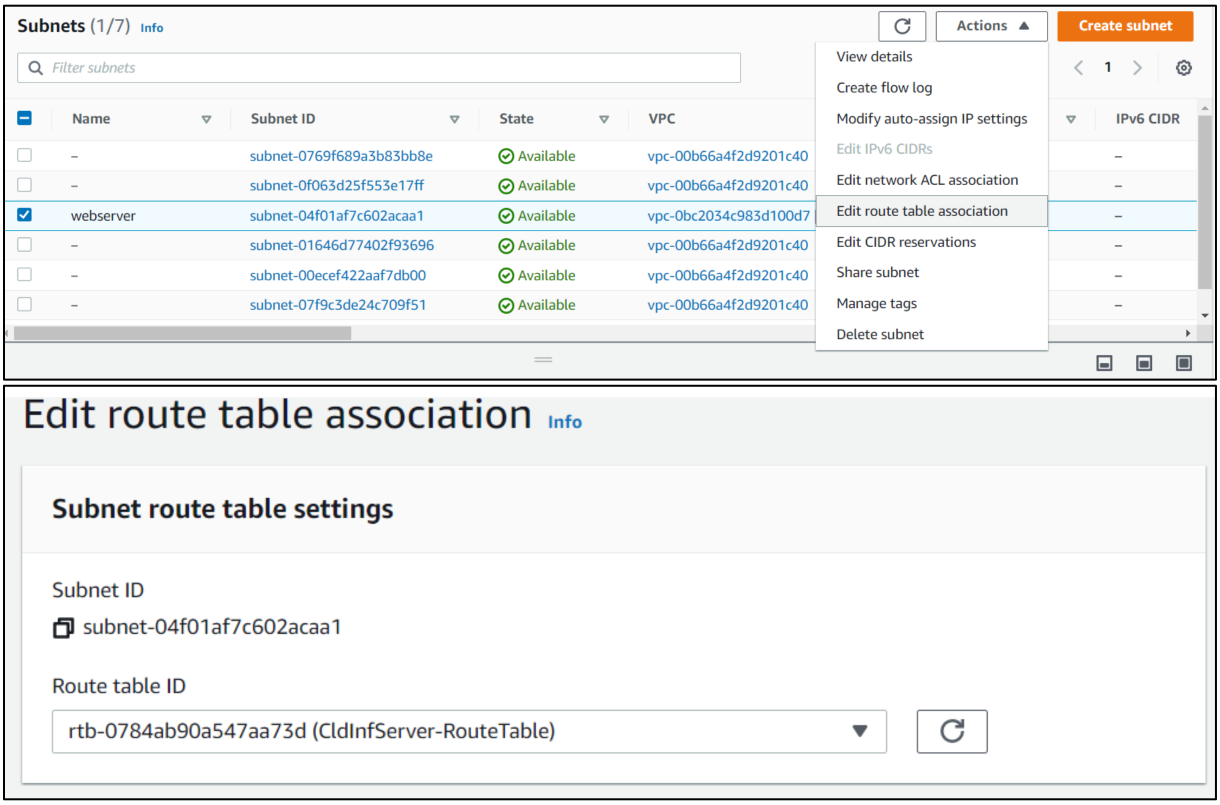


Figure 26 Editing and Associating Route Table

3.8 Installing Environment

This part consists of installing PostgreSQL and Nodejs with the given commands in the lab document and setup the Service Files.

For PostgreSQL the commands alone sufficed. But for Nodejs we first had to replace the “Environment” variables in the service file again using given commands for the lab document replacing following: AWS_REGION=us-east-1 (tta-api file) and GROUP_NAME=g07 (tta-web file).

Following the commands, we installed npm in the /opt/web folder and copied the file tta-api to /opt/.

Before starting the services, we had to edit the parameter store variables to match following:

```
The following variables have to be defined in the AWS Parameter Store:
|Parameter Store Variable|Description|Mandatory|
|---|---|---|---|
|/cldinf/dbUrl|Database Host IP or Domain Name|Yes|
|/cldinf/dbName|Database Name|Yes|
|/cldinf/dbUser|Database User|Yes|
|/cldinf/dbPassword|Database Password|Yes|
```

Figure 27 Parameter Store Variable Information

3.9 Adding IAM Role

To finish this section of the lab we had to add the pre-created IAM role to our Server. This role grants many AWS services access to other AWS services and has permissions very similar to the permissions we have as a user in the console. The name of the role is “LabInstanceProfile”. We simply added it to the EC2 instance under the security tab and then we could connect.

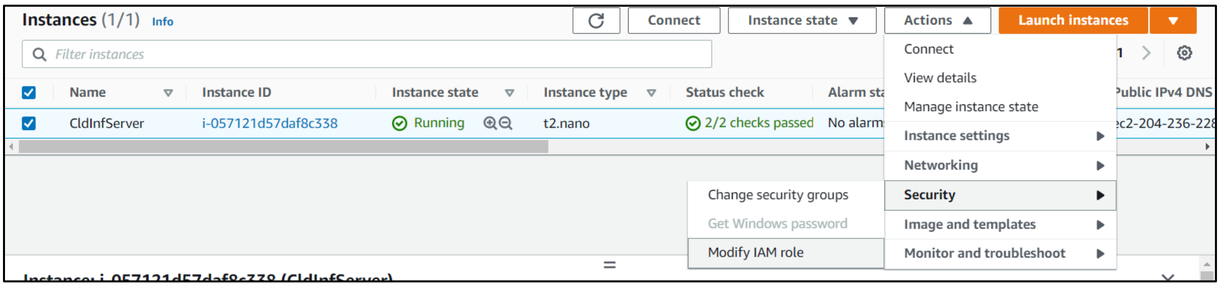


Figure 28 IAM role Security Tab

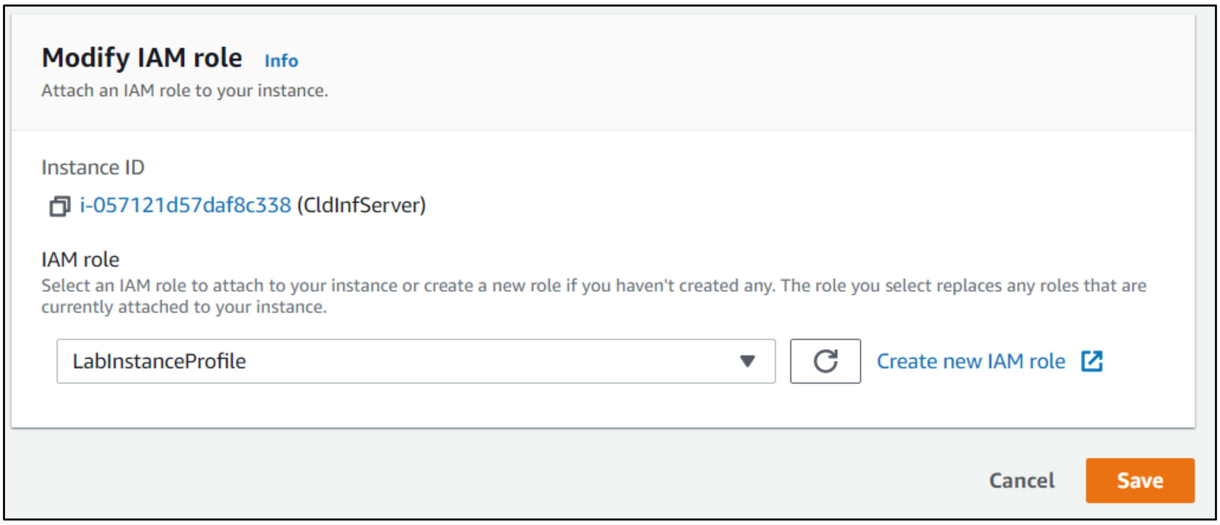


Figure 29 Adding IAM Role to instance

4 Migrating a Database to Amazon RDS

The database we use provides valuable information for the business we run it for, and we don't want to lose this information. This database should be durable, scalable, and work with high performance. Therefore, we migrate the database on our EC2 instance to an Amazon Relational Database Service (Amazon RDS). To be precise we transfer the already existing PostgreSQL database to the RDS.

The goal of this step was to create said database, import the data to it and connect an SQL client. Finally, we had to configure the EC2 instance to get its data from the RDS now instead of its own database.

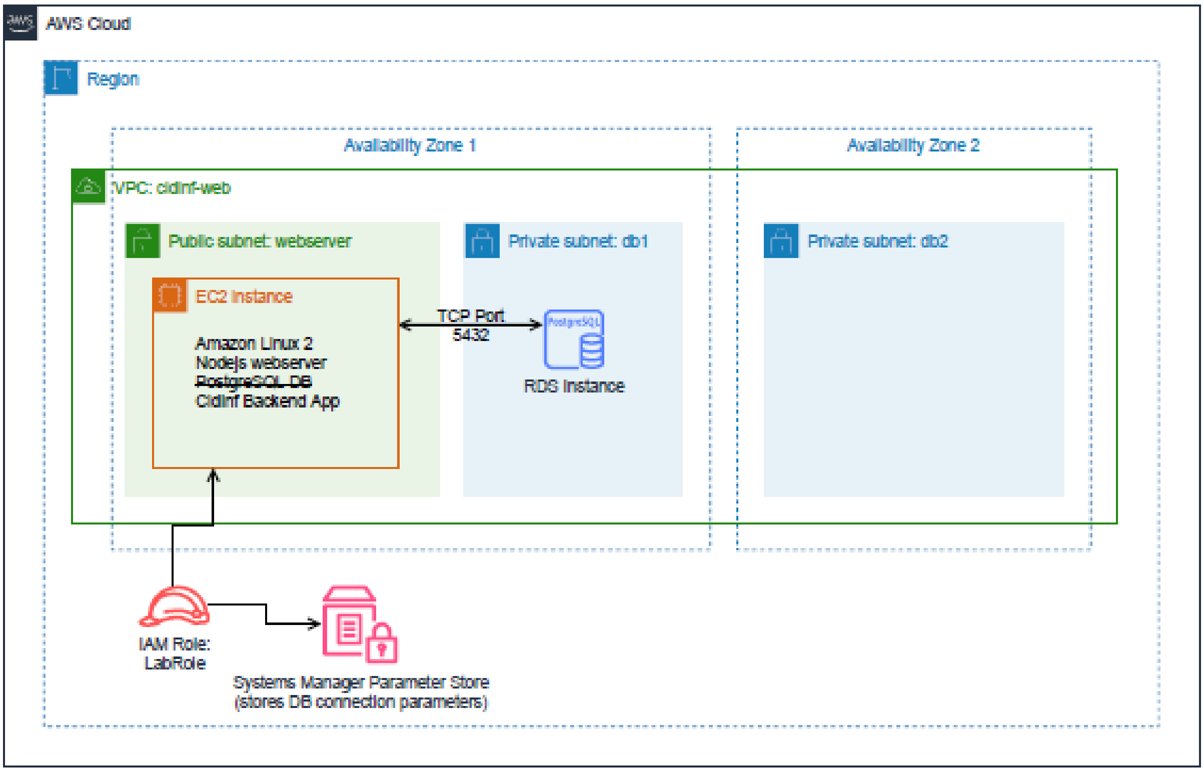


Figure 30 Example for a DB Migration to RDS instance

4.1 Creating the RDS Instance

4.1.1 Setting up RDS with given specifications

Similar to the rest of the lab we had to find the right tab to configure the database. This time it could be found under RDS/Create database. To have the best configurability we selected the "Standard create" option instead of the "Easy create" and for the Engine options to use the PostgreSQL. We named our database "cldinf-db".

Then we had to enter the credentials which were given to us in the lab (username and master password).

Next, we determined the DB instance class which decides the computation and memory capacity of our Amazon RDS DB instance and the allocated storage. These settings were also given by the lab document (db.t3.micro and 20GiB storage).

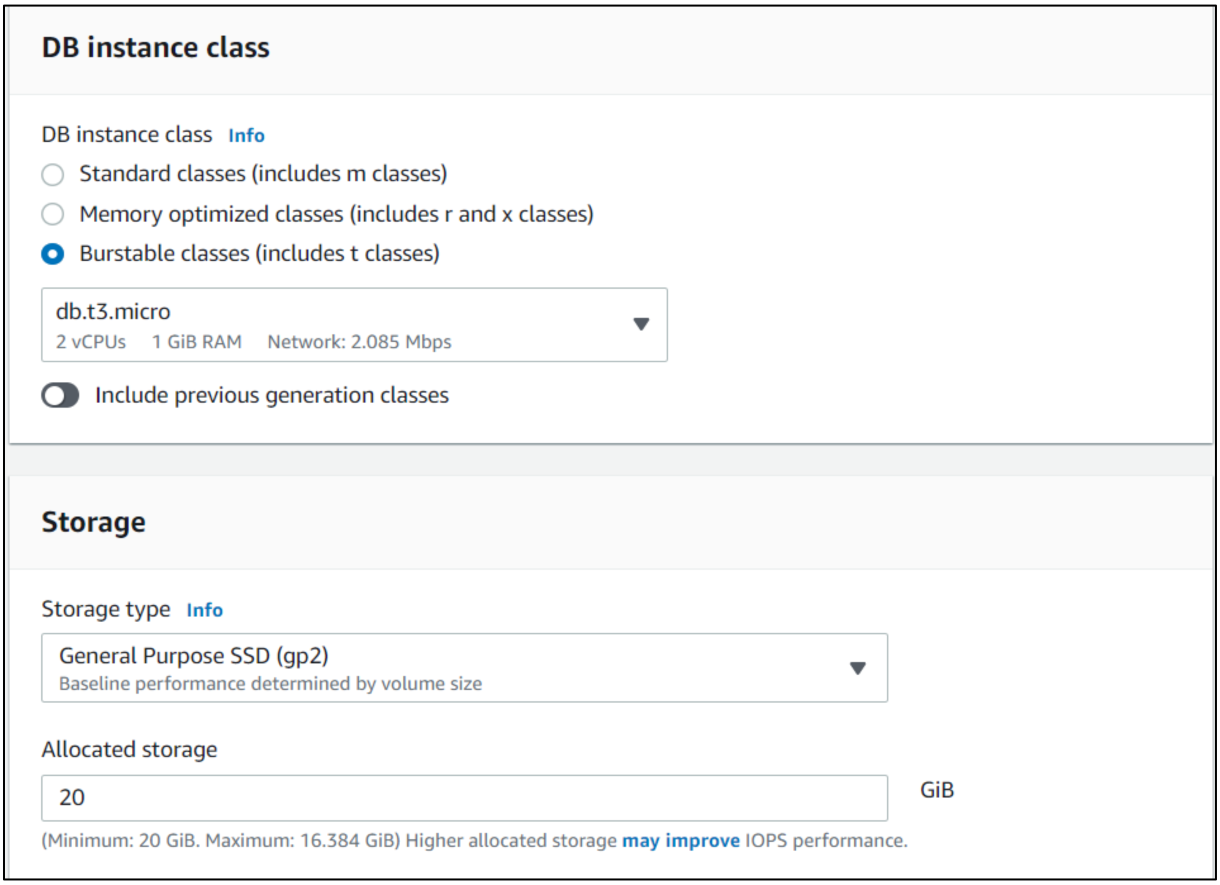
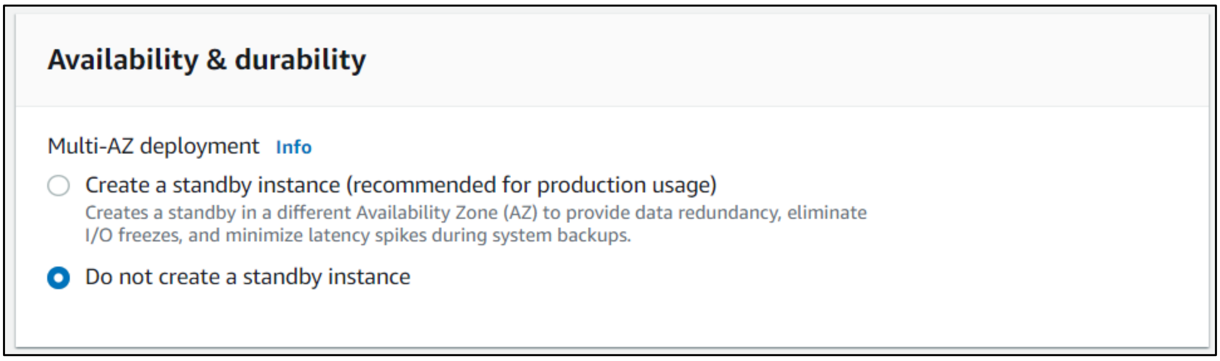


Figure 31 DB instance class and Storage

We didn't create a standby instance, which means that in the 2nd Availability Zone there won't be an exact copy of the database on standby. So, we basically don't have any redundancy, even though we will have to specify a 2nd subnet in another Availability Zone.



4.1.2 Setting up Connectivity

The RDS database needs to be placed in the same VPC as the EC2 instance (see figure 30). This can be configured in the connectivity tab of the instance. Additionally, we had to setup a subnet group for the database and add it to a VPC security group. The port to connect to could be specified in the "Additional Configuration" tab.

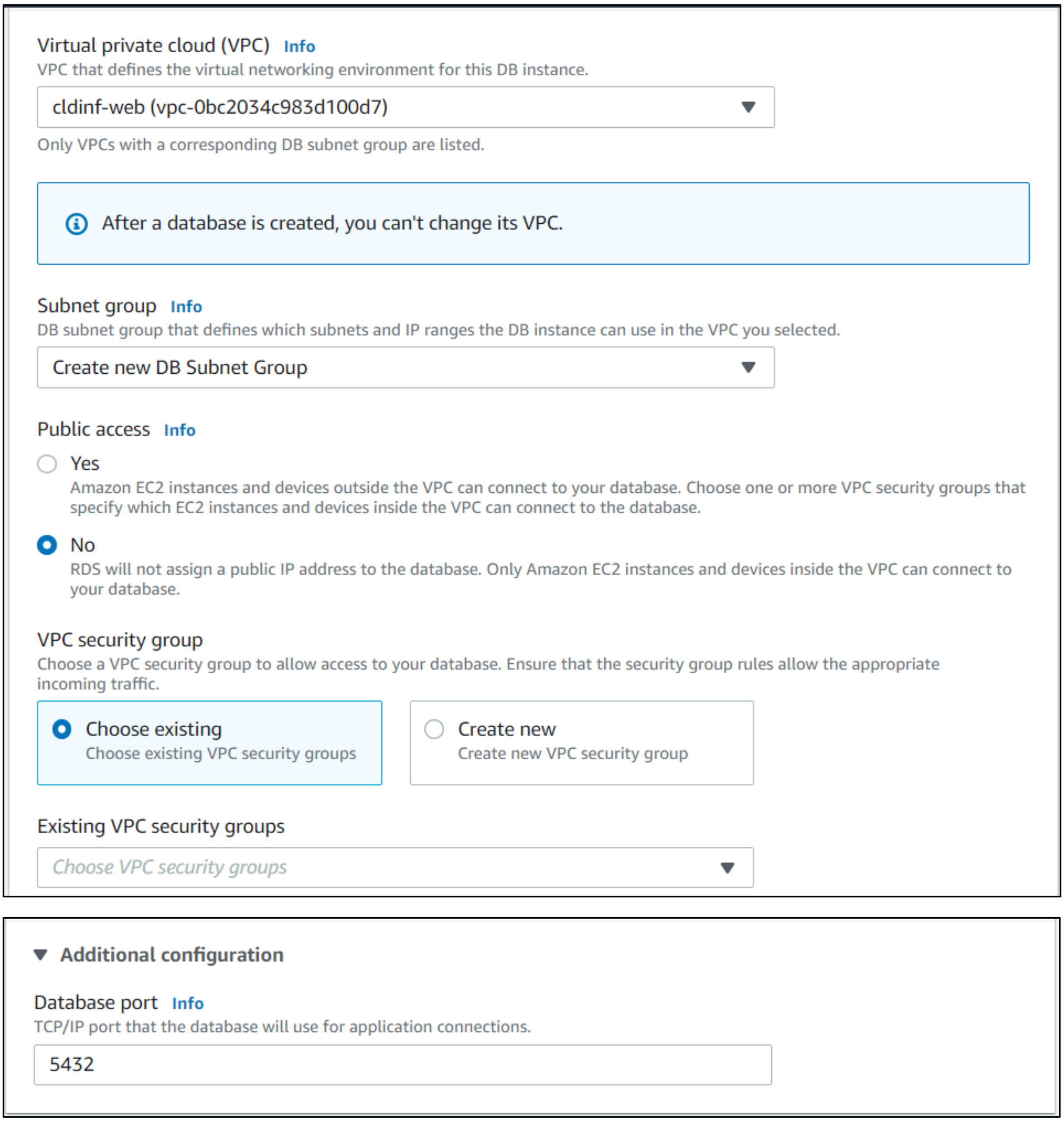


Figure 32 Add instance to VPC and select Port

4.1.3 Creating Subnet group

Looking at the example in figure 30 we can see that there are two database subnets (db1 and db2) in different availability zones to allow for a certain security, availability, and redundancy. These two subnets were created as we did before but using a different zone for each (us-east-1a and -1b). Both subnets were associated with the cldinf-web VPC.

1a was configured with the CIDR block of 172.32.16.0/24 while 1b uses 172.32.17.0/24 (figure 35).

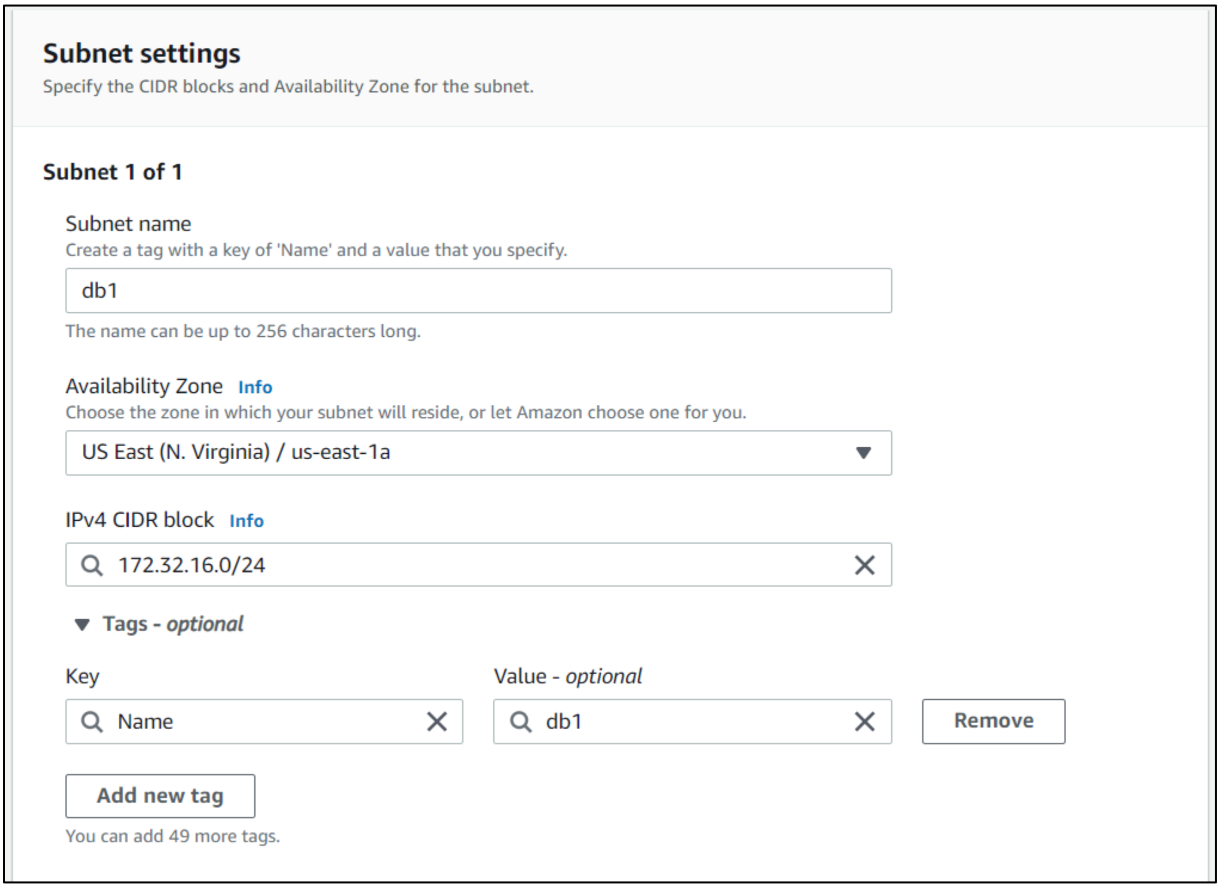


Figure 33 Setting up Subnet 1a (1b is similar but different CIDR block)

Then we just needed to edit the DB subnet group and add the two created subnets to it:

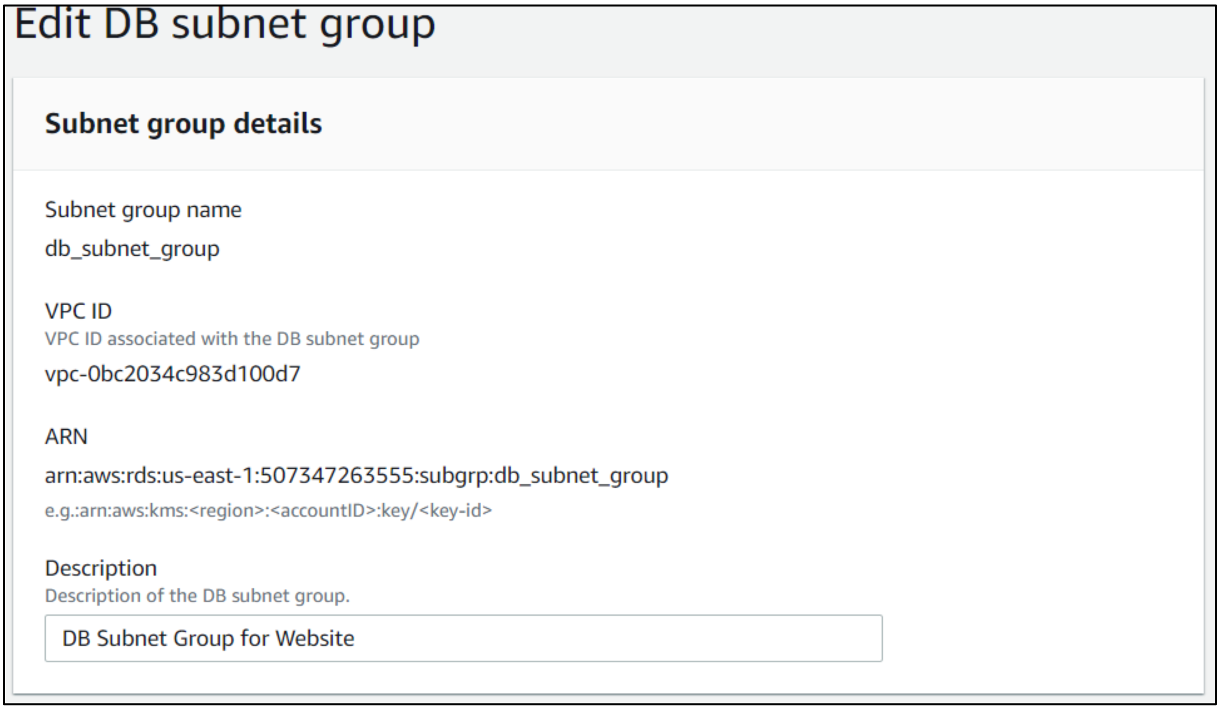


Figure 34 Edit DB subnet group

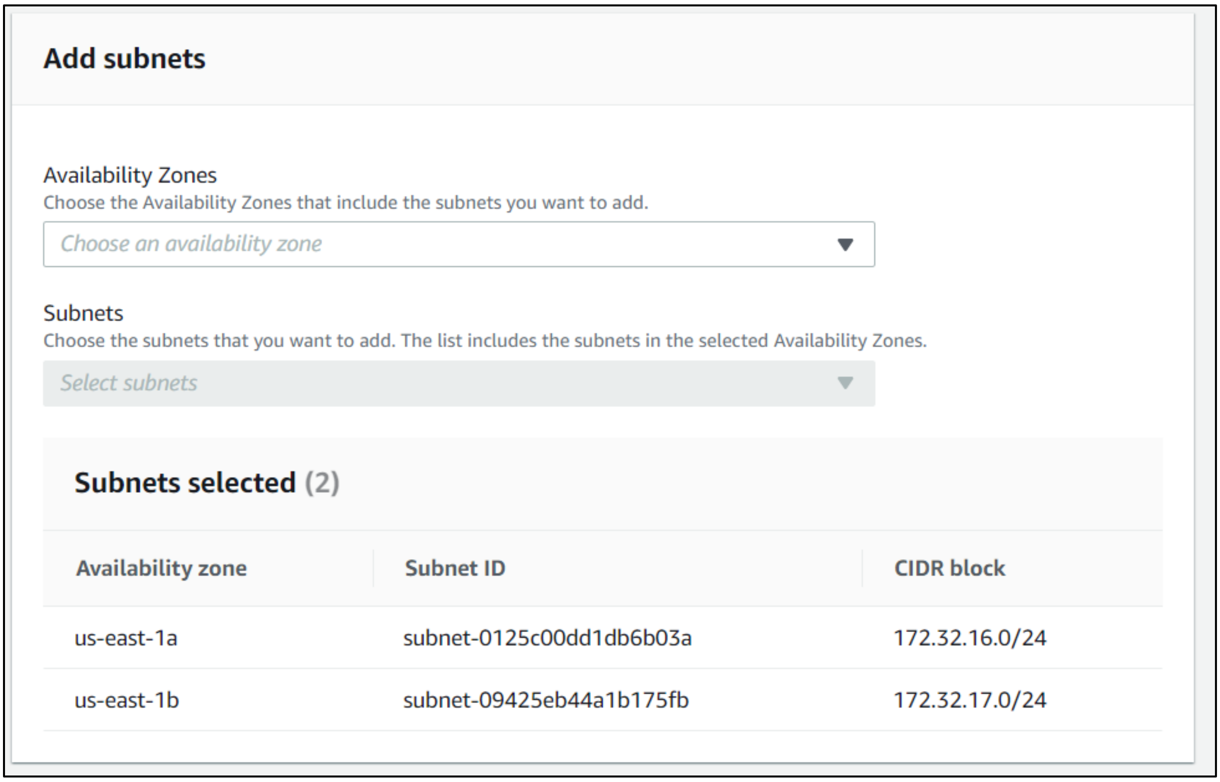


Figure 35 Adding Subnets 1a and 1b to the Group

Now that we have created this group, we could go back to the Connectivity options and select it in the dropdown menu:

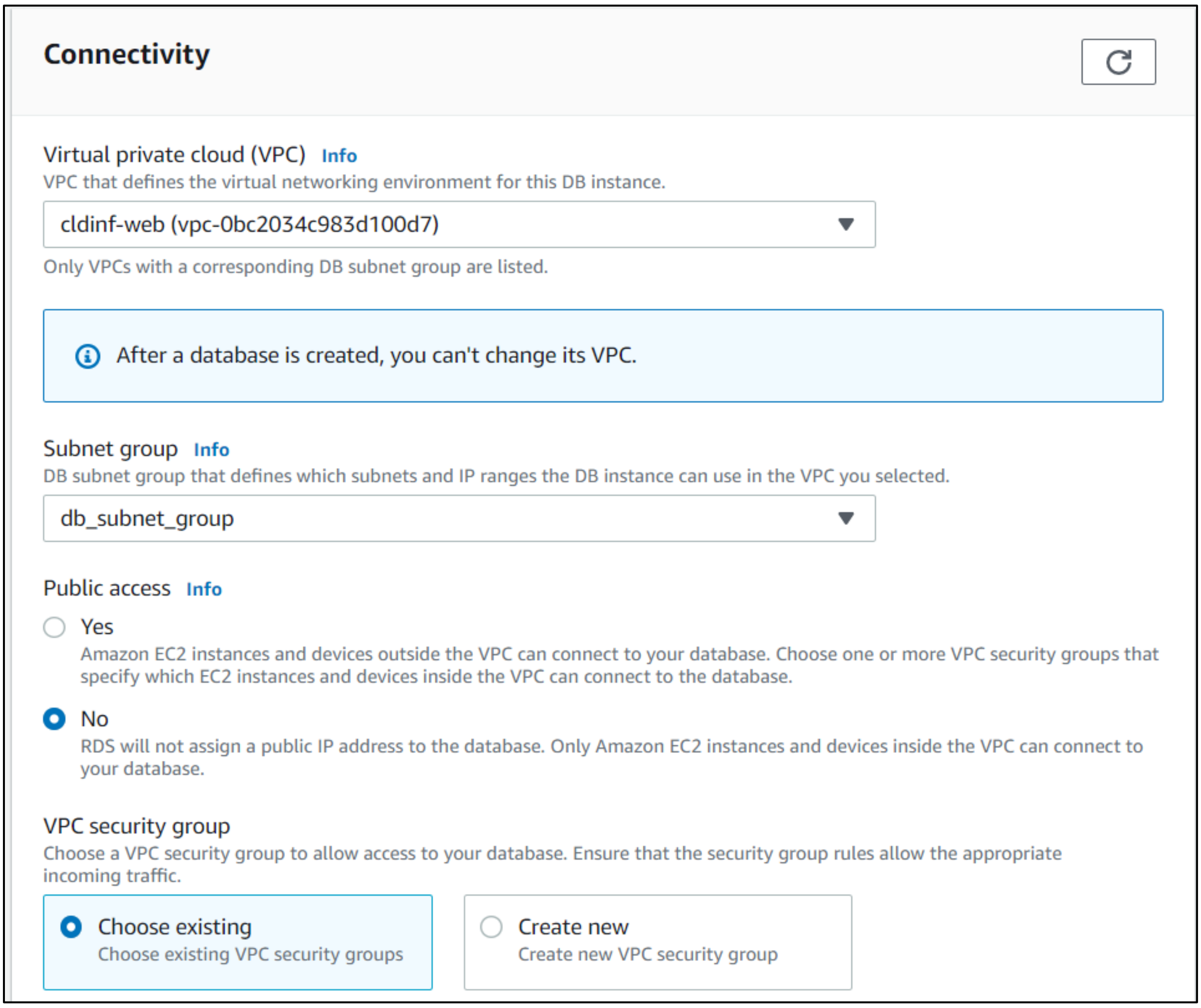


Figure 36 Final Connectivity Configurations

Now we just had to setup the initial name of the database and the DB parameter group. We disabled Monitoring, because else we couldn't have created the RDS instance, since we don't have the rights in the learner labs to activate "Enhanced Monitoring".

4.2 Verify EC2 Database and Exporting Data

To verify the data on the EC2 database we had to enter some commands: `psql --version` to check if the version running on the instance is posgresql-13. Then we connected to the data-base by entering: `psql -U postgres -h localhost postgres`. To observe the current data on the database we could enter a series of commands that show all the log events that were created.

```
postgres=# Select * from logs;
 id |     content      |          created_at
----+------------------+-------------------------------
  1 | localhost:8000 | 2021-11-16 12:51:18.874467+00
  2 | localhost:8000 | 2021-11-16 12:56:36.985507+00
  3 | localhost:8000 | 2021-11-16 12:56:37.387571+00
  4 | localhost:8000 | 2021-11-16 12:58:34.925101+00
  5 | localhost:8000 | 2021-11-16 12:58:39.095045+00
(5 rows)
```

Figure 37 Postgres Log Event data on EC2 instance

To export the data in this table we used the pg_dump utility: `pg_dump -U postgres -h <localhost> -F t postgres > /tmp/cldinf-db.tar` and to check if the dump was successful, we could look at the /tmp folder to see if the .tar file we dumped into was created.

4.3 Importing Data to new RDS Database

The last step before we could complete this part of the lab was to import the created .tar files data into the new RDS database. To successfully do that we had to first establish a network connection from the terminal running on the EC2 instance to the new RDS instance by updating the inbound rules of the security group that the RDS instance runs in. The port we allowed to connect was the before defined port 5432. To not create a security risk, we only opened the connection to servers in the security group that is used by the EC2 instance were connecting from.

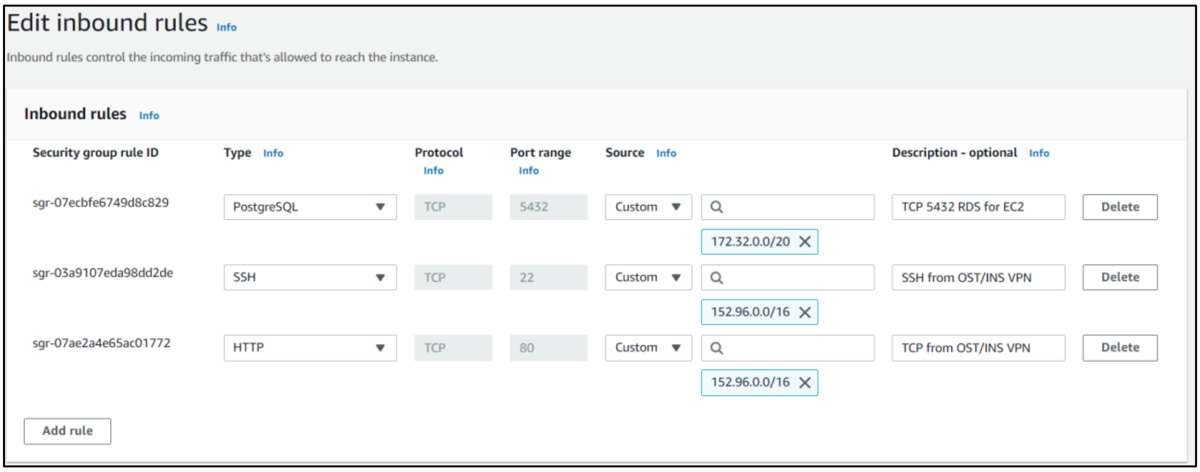


Figure 38 Updating Security Group of the RDS Instance

Then we connect to the RDS instance from our EC2 instance by entering: `psql -U postgres -h <rds-instance> postgres`. Run the `\l` command to show the databases and list all the tables of the "postgres" DB. Obviously there weren't any tables at the time we did this, so we imported our data from the EC2 instance .tar with: `pg_restore -d postgres /tmp/cldinf-db.tar -c -U postgres -h <rds-instance>`. And check again if the data exists in the database with the commands we used before and were given us by the lab document.

```
[ec2-user@ip-172-32-7-61 ~]$ psql -U postgres -h cldinf-db.czmdhm4ch9uu.us-east-1.rds.amazonaws.com postgres
Password for user postgres:
psql (13.4, server 13.3)
SSL connection (protocol: TLSv1.2, cipher: ECDHE-RSA-AES256-GCM-SHA384, bits: 256, compression: off)
Type "help" for help.

postgres=> \l
                                  List of databases
   Name    |  Owner   | Encoding |   Collate   |    Ctype    |   Access privileges
-----------+----------+----------+-------------+-------------+-----------------------
 postgres  | postgres | UTF8     | en_US.UTF-8 | en_US.UTF-8 |
 rdsadmin  | rdsadmin | UTF8     | en_US.UTF-8 | en_US.UTF-8 | rdsadmin=CTc/rdsadmin
 template0 | rdsadmin | UTF8     | en_US.UTF-8 | en_US.UTF-8 | =c/rdsadmin          +
           |          |          |             |             | rdsadmin=CTc/rdsadmin
 template1 | postgres | UTF8     | en_US.UTF-8 | en_US.UTF-8 | =c/postgres          +
           |          |          |             |             | postgres=CTc/postgres
(4 rows)

postgres=> \ postgres
invalid command \
Try \? for help.
postgres=> \c postgres
psql (13.4, server 13.3)
SSL connection (protocol: TLSv1.2, cipher: ECDHE-RSA-AES256-GCM-SHA384, bits: 256, compression: off)
You are now connected to database "postgres" as user "postgres".
postgres=> \d
             List of relations
 Schema |     Name     |   Type   |  Owner
--------+--------------+----------+----------
 public | logs         | table    | postgres
 public | logs_id_seq  | sequence | postgres
(2 rows)

postgres=> select * from logs;
 id |    content     |          created_at
----+----------------+-------------------------------
  1 | localhost:8000 | 2021-11-16 12:51:18.874467+00
  2 | localhost:8000 | 2021-11-16 12:56:36.985507+00
  3 | localhost:8000 | 2021-11-16 12:56:37.387571+00
  4 | localhost:8000 | 2021-11-16 12:58:34.925101+00
  5 | localhost:8000 | 2021-11-16 12:58:39.095045+00
(5 rows)
```

4.4 Confirming that Application still works

To confirm this, we had to connect the cldinf application to the new database and stop the database that runs locally on the EC2 instance by first updating the necessary values in the Parameter Store (dbUrl).

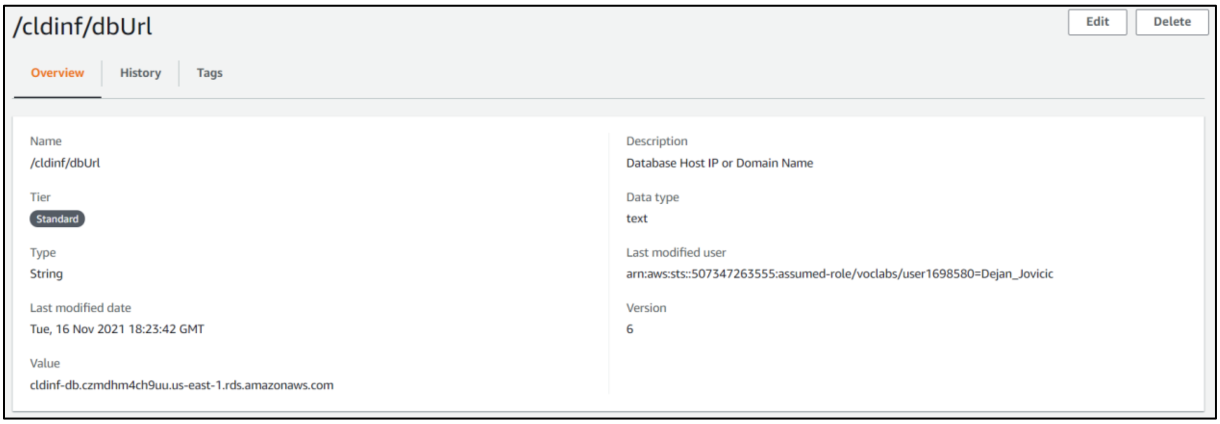


Figure 39 Changing dbURL Parameter Store Value to connect EC2 instance to new database

After changing this value, we could stop the running database on the EC2 instance and load the page again to confirm the data being taken from the right database.

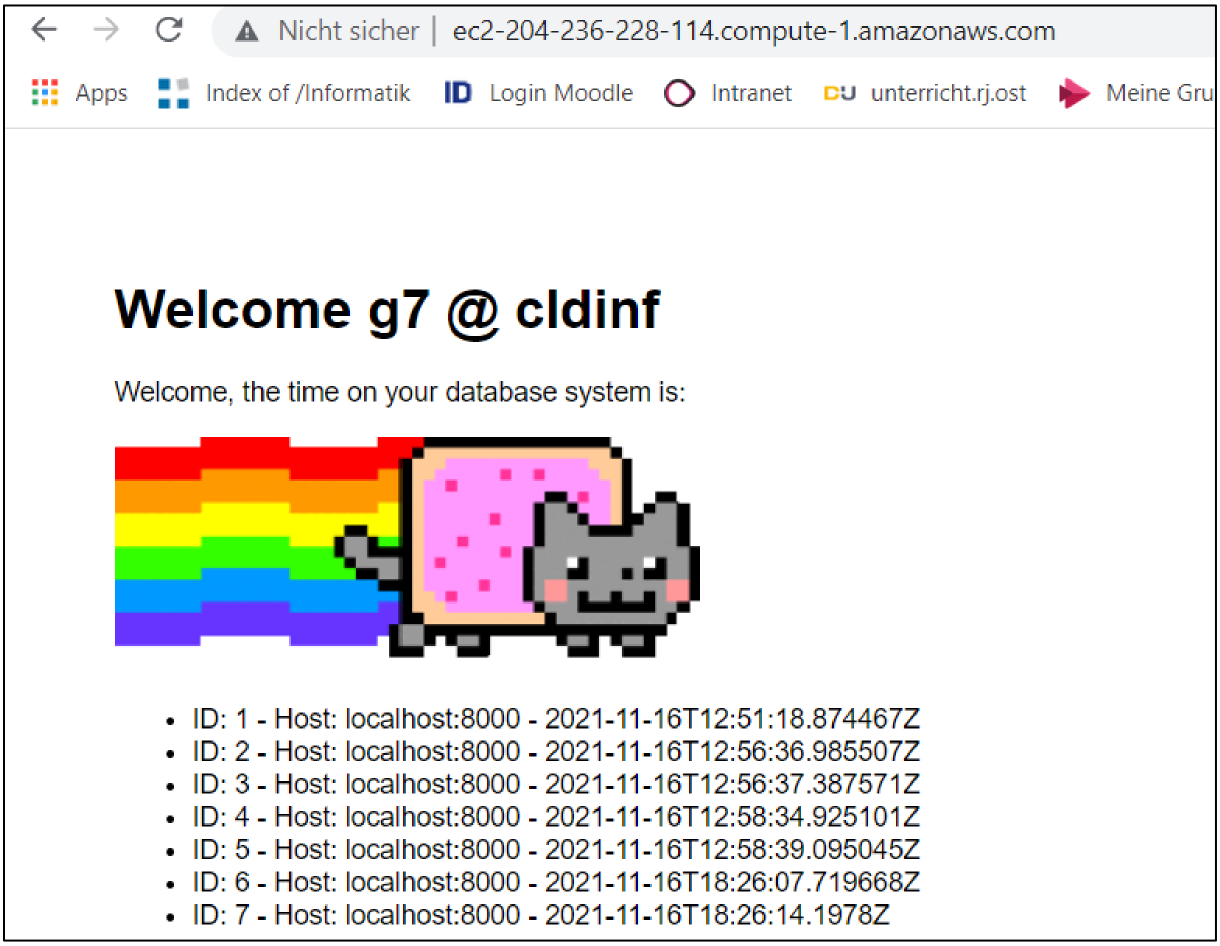


Figure 40 Confirmation that the EC2 instance takes the data from the correct database

5 Implementing a scalable and highly available environment

First of all, we are going to create an Auto Scaling group named “autoscaling_cldinf”. A launch template is required and therefore that is the next step.

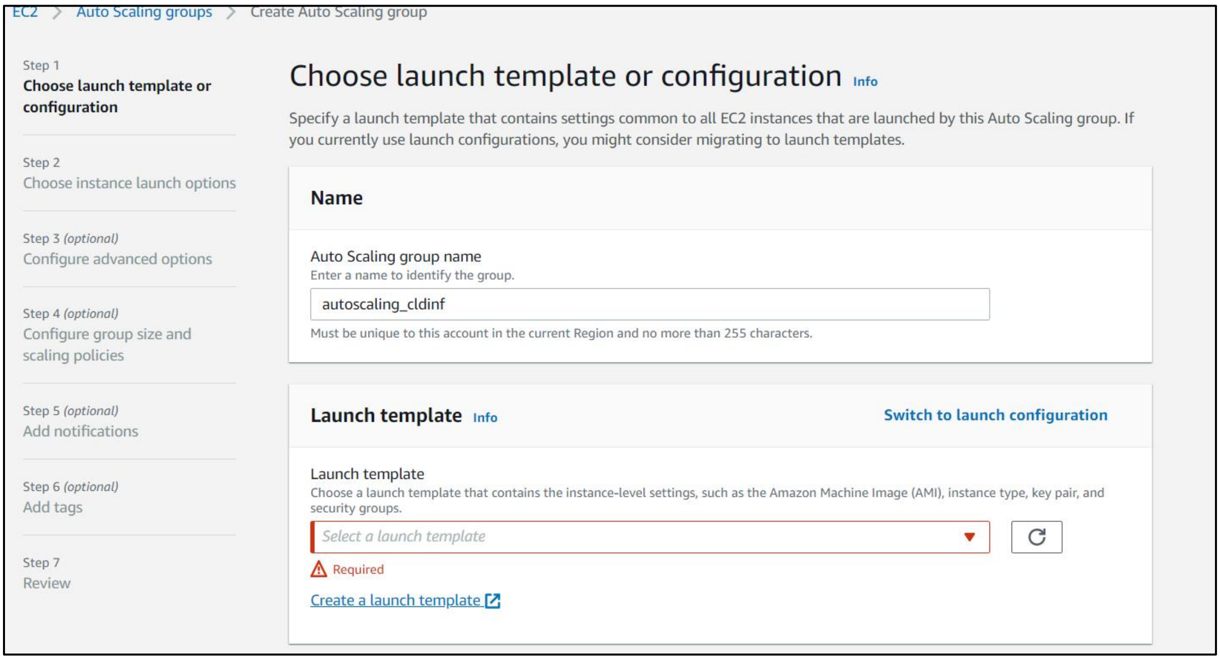


Figure 41 Creating Autoscaling Group

With the launch template, we can specify how the auto provisioned VMs will look like. We can specify the security group, in what VPC the VMs should be, IAM instance profiles, etc.

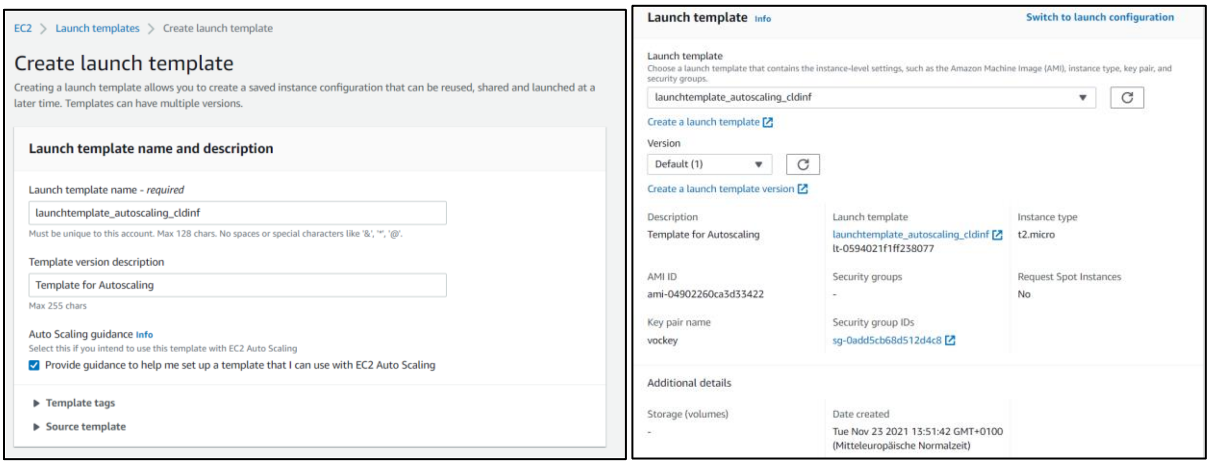


Figure 42 Creating Launch Template

After we've done this, we decided to attach our Auto Scaling group to a new load balancer, since we've haven't created a load balancer yet. The load balancer is an "Application Load Balancer", and the scheme should be "Internet-facing".

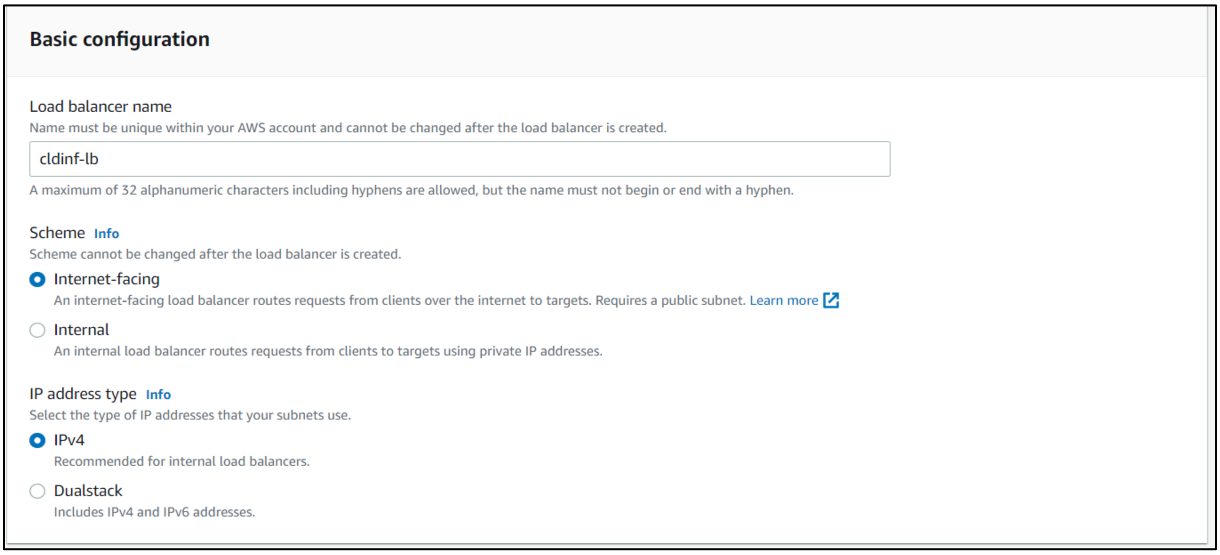


Figure 43 Configuring Load Balancer

We add the two private subnets to the load balancer and create a new target group name for the listener.

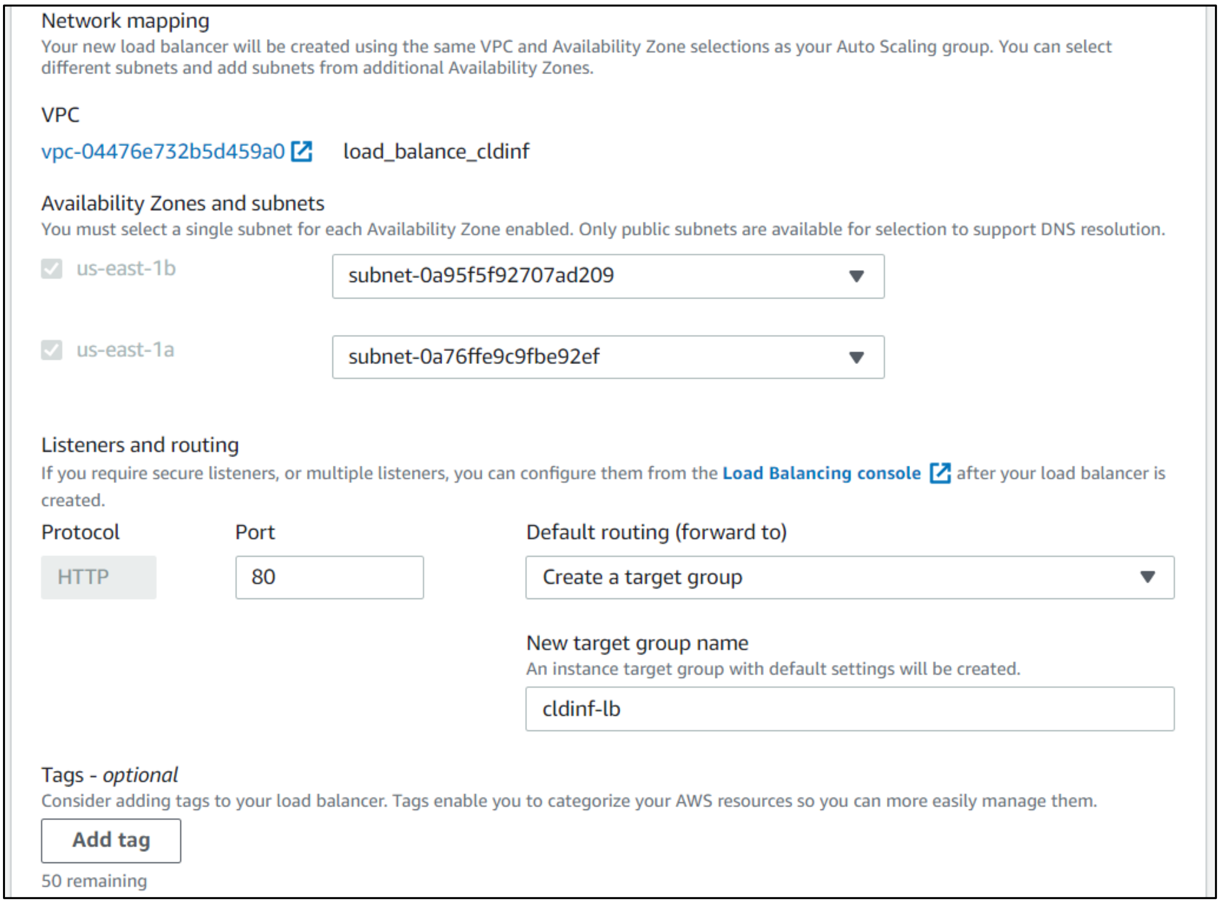


Figure 44 Adding Subnets to Load Balancer

After we've done this, we set the "Desired Capacity" and "Minimum capacity" to 2 and the "Maximum capacity" to 6 on the Auto Scaling group. So, the minimum number of VMs should be 2, the maximum number 6. We also have to set the scaling policies as shown in this screenshot.

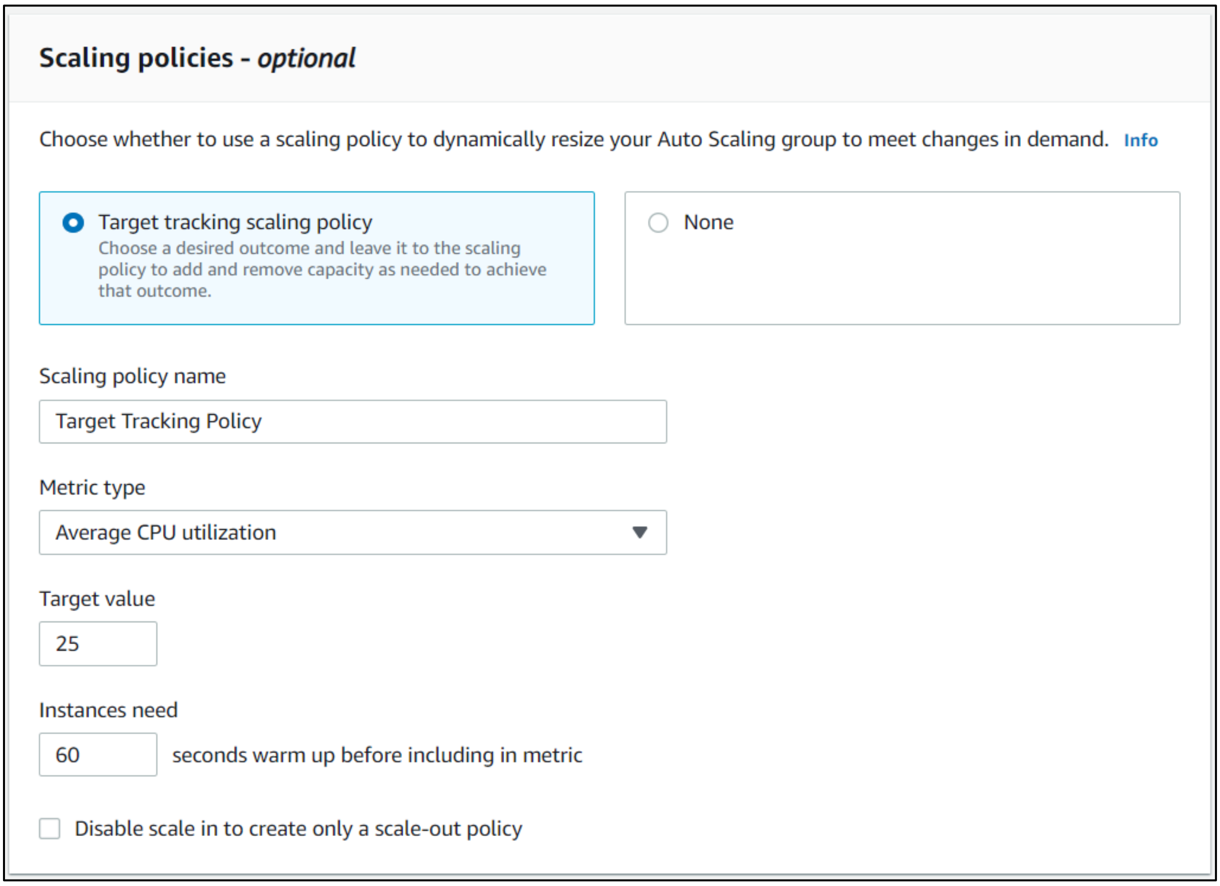


Figure 45 Setting up Scaling Policies

5.1 Connection Setup for the Load Balancer and Autoscaled VMs

We have tried to construct this network, as shown in the Powerpoint presentation of Prof. Laurent Metzger.

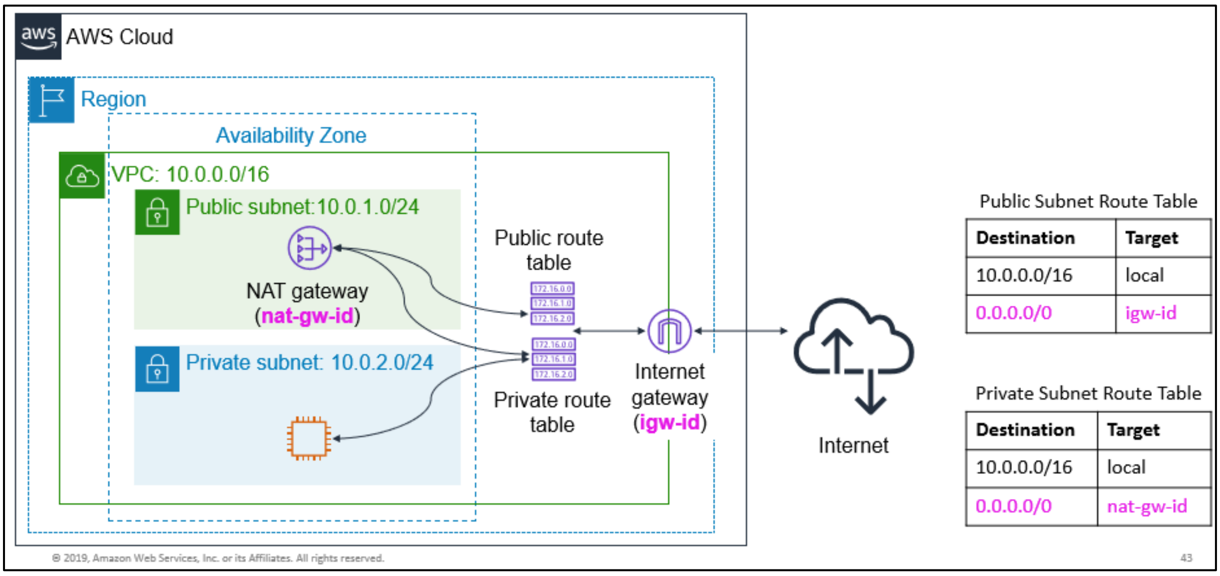


Figure 46 Source: Lecture "05_AWS part 1" slide 43

The first step was to create a NAT gateway, in the public subnet (172.33.18.0./24) we've specifically created for this case.

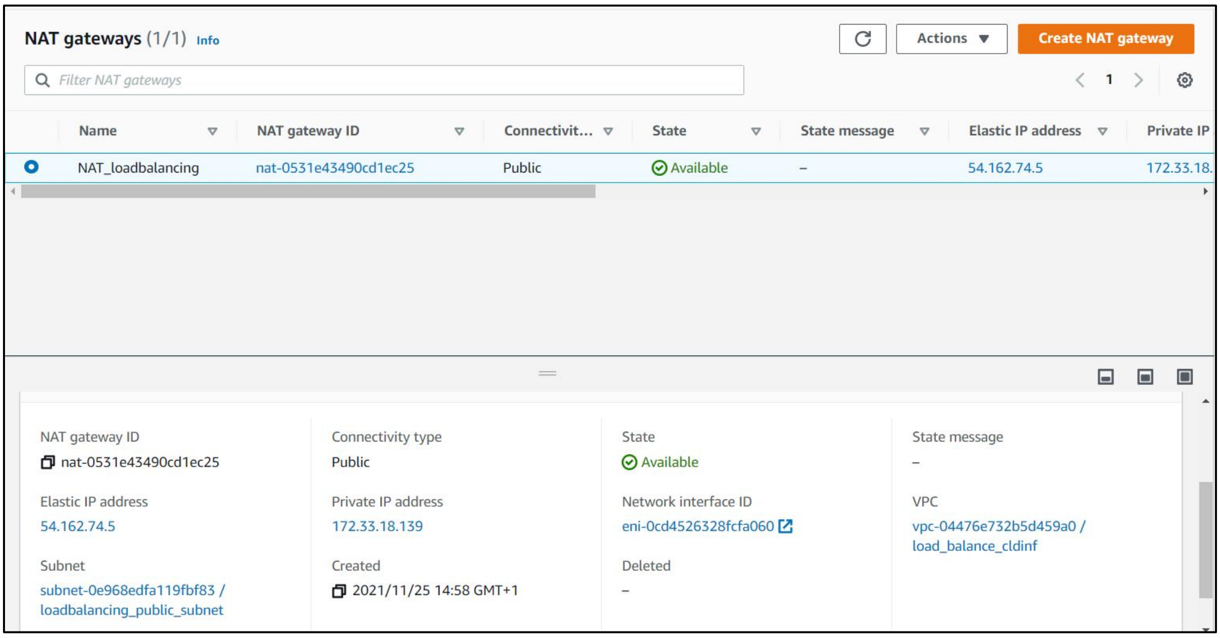


Figure 47 NAT gateway UI

Then we created a route table for our NAT connection, add route to 0.0.0.0/0 via the created NAT. The VMs in the subnets 172.33.16.0/24 and 172.33.17.0/24 should now connect to the Internet using the NAT Gateway. You can also see a route for the peering of two different VPCs. This was configured so we could connect from the EC2 instance we've previously created in the lab, to the newly created VMs in the Auto Scaling group. We won’t go into further detail, because this wasn’t a part of the lab, and we did it only for our convenience.

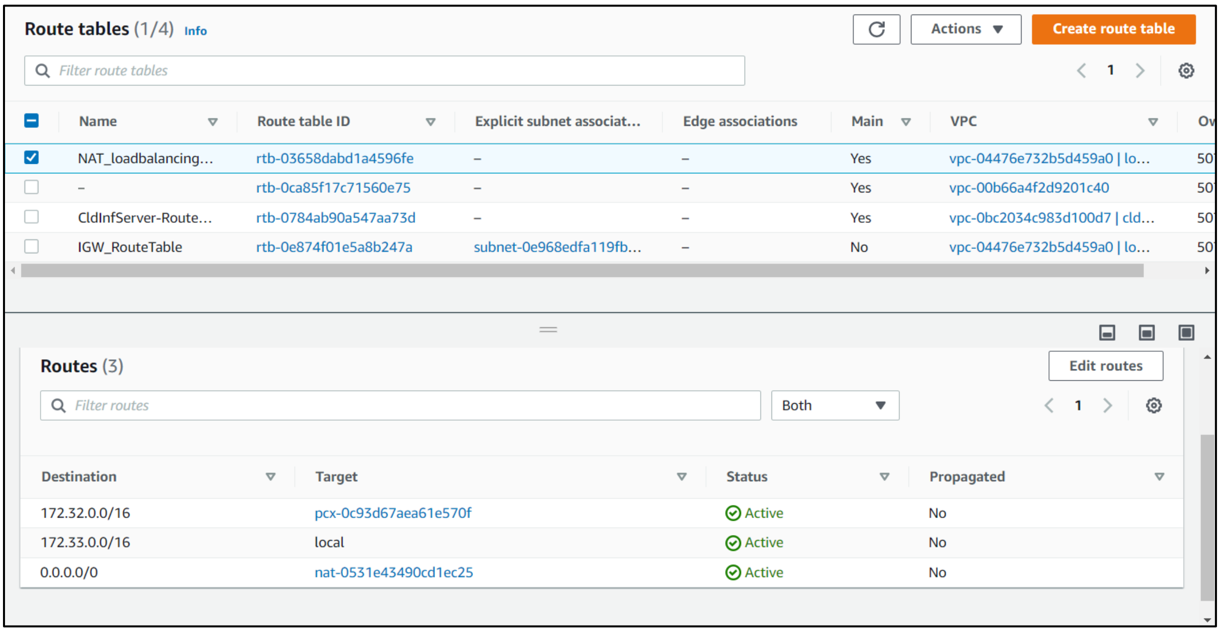


Figure 48 Creating NAT Route Table

We also had to create a route table for the Internet Gateway and add explicitly the public subnet 172.33.18.0/24. This must be done, since the IGW_RouteTable isn’t a main route table and therefore subnets have to be added explicitly.

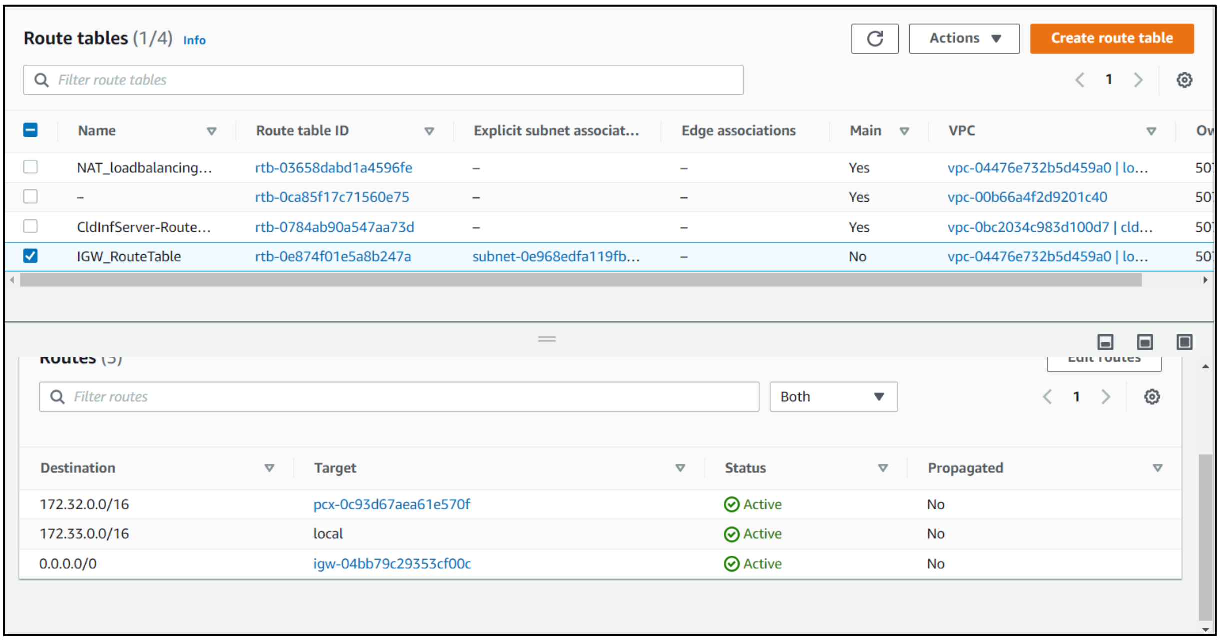
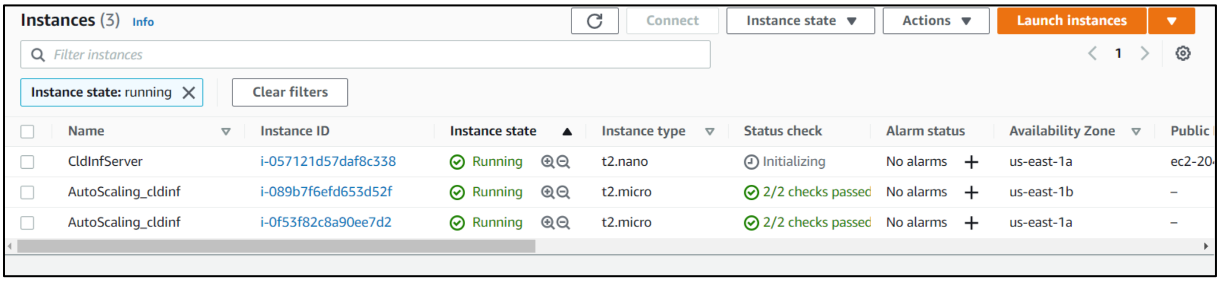


Figure 49 Internet Gateway for the Route Table

Sadly, this network design didn’t work properly, as either the VM’s in the private subnets 172.33.16.0/24 and 172.33.17.0/24 had internet connection, but then you couldn’t have access to their “websites” through the load balancer or vice versa. If we added the private subnets to the IGW Route Table, the load balancer was working fine, but the VMs didn't have any internet connection and therefore couldn't install NGINX. If we added the private subnets to the NAT Route Table, the VMs had internet connection, but you couldn't connect to them via the load balancer. This kind of made sense, since when you have a NAT, you shouldn't be able to connect from outside to inside. In the end, we didn't know what was missing in our setup and decided to give every VM a public IP, by changing the auto launch template, and put the private subnets in the IGW Route Table.

5.2 Autoscaling

Started the AWS Lab and we had instantly 2 VMs named "Autoscaling_cldinf" getting provisioned.



Instances (3) Info | Connect | Instance state | Actions | Launch instances

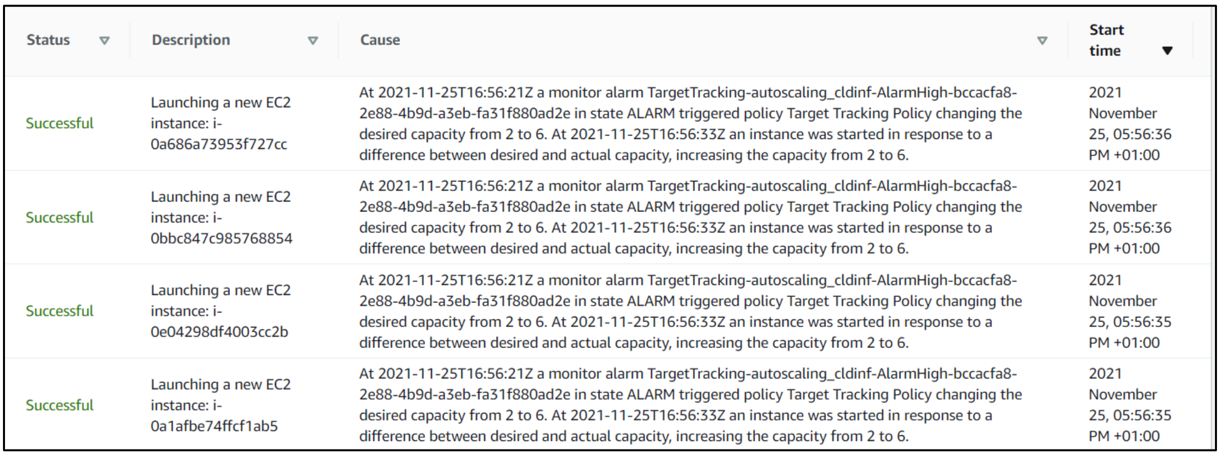
Filter instances

Instance state: running | Clear filters

Name	Instance ID	Instance state	Instance type	Status check	Alarm status	Availability Zone	Public
CldInfServer	i-057121d57daf8c338	Running	t2.nano	Initializing	No alarms	us-east-1a	ec2-20-
AutoScaling_cldinf	i-089b7f6efd653d52f	Running	t2.micro	2/2 checks passed	No alarms	us-east-1b	–
AutoScaling_cldinf	i-0f53f82c8a90ee7d2	Running	t2.micro	2/2 checks passed	No alarms	us-east-1a	–

Figure 50 Example for Autoscaling

After starting the stress test, we can see in the activity register, that the autoscaling works perfectly.



Status	Description	Cause	Start time
Successful	Launching a new EC2 instance: i-0a686a73953f727cc	At 2021-11-25T16:56:21Z a monitor alarm TargetTracking-autoscaling_cldinf-AlarmHigh-bccacfa8-2e88-4b9d-a3eb-fa31f880ad2e in state ALARM triggered policy Target Tracking Policy changing the desired capacity from 2 to 6. At 2021-11-25T16:56:33Z an instance was started in response to a difference between desired and actual capacity, increasing the capacity from 2 to 6.	2021 November 25, 05:56:36 PM +01:00
Successful	Launching a new EC2 instance: i-0bbc847c985768854	At 2021-11-25T16:56:21Z a monitor alarm TargetTracking-autoscaling_cldinf-AlarmHigh-bccacfa8-2e88-4b9d-a3eb-fa31f880ad2e in state ALARM triggered policy Target Tracking Policy changing the desired capacity from 2 to 6. At 2021-11-25T16:56:33Z an instance was started in response to a difference between desired and actual capacity, increasing the capacity from 2 to 6.	2021 November 25, 05:56:36 PM +01:00
Successful	Launching a new EC2 instance: i-0e04298df4003cc2b	At 2021-11-25T16:56:21Z a monitor alarm TargetTracking-autoscaling_cldinf-AlarmHigh-bccacfa8-2e88-4b9d-a3eb-fa31f880ad2e in state ALARM triggered policy Target Tracking Policy changing the desired capacity from 2 to 6. At 2021-11-25T16:56:33Z an instance was started in response to a difference between desired and actual capacity, increasing the capacity from 2 to 6.	2021 November 25, 05:56:35 PM +01:00
Successful	Launching a new EC2 instance: i-0a1afbe74ffcf1ab5	At 2021-11-25T16:56:21Z a monitor alarm TargetTracking-autoscaling_cldinf-AlarmHigh-bccacfa8-2e88-4b9d-a3eb-fa31f880ad2e in state ALARM triggered policy Target Tracking Policy changing the desired capacity from 2 to 6. At 2021-11-25T16:56:33Z an instance was started in response to a difference between desired and actual capacity, increasing the capacity from 2 to 6.	2021 November 25, 05:56:35 PM +01:00

Figure 51 Activity Register to Check Autoscaling

There are now 6 “autoscaling_cldinf” VMs.

	Name	Instance ID	Instance state	Instance type	Status check	Alarm status	Availability Zone	Public
☐	CldInfServer	i-057121d57daf8c338	Running	t2.nano	2/2 checks passed	No alarms	us-east-1a	ec2-20
☐	AutoScaling_cldinf	i-0db3e65ec997c671e	Running	t2.micro	2/2 checks passed	No alarms	us-east-1b	–
☐	–	i-0c00ef4699e73a833	Running	t2.micro	2/2 checks passed	No alarms	us-east-1b	–
☑	AutoScaling_cldinf	i-0e04298df4003cc2b	Running	t2.micro	2/2 checks passed	No alarms	us-east-1b	–
☐	AutoScaling_cldinf	i-0a1afbe74ffcf1ab5	Running	t2.micro	2/2 checks passed	No alarms	us-east-1b	–
☐	AutoScaling_cldinf	i-011b8202dfa1fd7f4	Running	t2.micro	2/2 checks passed	No alarms	us-east-1a	–
☐	AutoScaling_cldinf	i-0bbc847c985768854	Running	t2.micro	2/2 checks passed	No alarms	us-east-1a	–
☐	AutoScaling_cldinf	i-0a686a73953f727cc	Running	t2.micro	2/2 checks passed	No alarms	us-east-1a	–

Figure 52 Autoscaling after Test

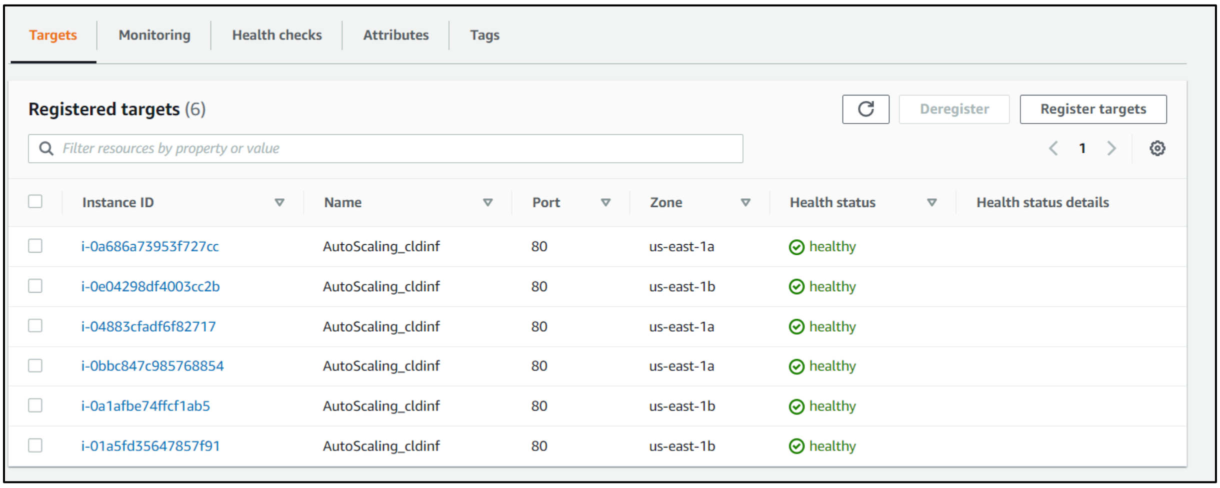
We can also wait some time and see that the instances will get terminated again, after the CPU usage has been reduced.

Successful	Terminating EC2 instance: i-0bbc847c985768854	At 2021-11-25T17:25:23Z a monitor alarm TargetTracking-autoscaling_cldinf-AlarmLow-d5968f4c-e624-4a83-8c2a-5865b3b0921a in state ALARM triggered policy Target Tracking Policy changing the desired capacity from 3 to 2. At 2021-11-25T17:25:32Z an instance was taken out of service in response to a difference between desired and actual capacity, shrinking the capacity from 3 to 2. At 2021-11-25T17:25:32Z instance i-0bbc847c985768854 was selected for termination.	2021 November 25, 06:25:32 PM +01:00
Successful	Terminating EC2 instance: i-0e04298df4003cc2b	At 2021-11-25T17:23:45Z a monitor alarm TargetTracking-autoscaling_cldinf-AlarmLow-2831714b-9cea-45c1-83c3-434e17b13706 in state ALARM triggered policy Target Tracking Policy changing the desired capacity from 4 to 3. At 2021-11-25T17:23:55Z an instance was taken out of service in response to a difference between desired and actual capacity, shrinking the capacity from 4 to 3. At 2021-11-25T17:23:56Z instance i-0e04298df4003cc2b was selected for termination.	2021 November 25, 06:23:56 PM +01:00
Successful	Terminating EC2 instance: i-0a686a73953f727cc	At 2021-11-25T17:22:27Z a monitor alarm TargetTracking-autoscaling_cldinf-AlarmLow-b7182220-40ab-4c7b-bdb4-a942a1c4dadf in state ALARM triggered policy Target Tracking Policy changing the desired capacity from 5 to 4. At 2021-11-25T17:22:38Z an instance was taken out of service in response to a difference between desired and actual capacity, shrinking the capacity from 5 to 4. At 2021-11-25T17:22:38Z instance i-0a686a73953f727cc was selected for termination.	2021 November 25, 06:22:38 PM +01:00
Successful	Terminating EC2 instance: i-0a1afbe74ffcf1ab5	At 2021-11-25T17:21:27Z a monitor alarm TargetTracking-autoscaling_cldinf-AlarmLow-b7182220-40ab-4c7b-bdb4-a942a1c4dadf in state ALARM triggered policy Target Tracking Policy changing the desired capacity from 6 to 5. At 2021-11-25T17:21:40Z an instance was taken out of service in response to a difference between desired and actual capacity, shrinking the capacity from 6 to 5. At 2021-11-25T17:21:40Z instance i-0a1afbe74ffcf1ab5 was selected for termination.	2021 November 25, 06:21:40 PM +01:00

Figure 53 Activity Register to Check Autoscaling after Test

5.3 Load Balancing

We can see that all clients are healthy.



Targets Monitoring Health checks Attributes Tags

Registered targets (6)

Deregister Register targets

Filter resources by property or value

Instance ID	Name	Port	Zone	Health status	Health status details
i-0a686a73953f727cc	AutoScaling_cldinf	80	us-east-1a	healthy	
i-0e04298df4003cc2b	AutoScaling_cldinf	80	us-east-1b	healthy	
i-04883cfadf6f82717	AutoScaling_cldinf	80	us-east-1a	healthy	
i-0bbc847c985768854	AutoScaling_cldinf	80	us-east-1a	healthy	
i-0a1afbe74ffcf1ab5	AutoScaling_cldinf	80	us-east-1b	healthy	
i-01a5fd35647857f91	AutoScaling_cldinf	80	us-east-1b	healthy	

Figure 54 "Healthy"-Check for the Clients

And when accessing the load balancer, we can see that it goes round robin through all the instances:

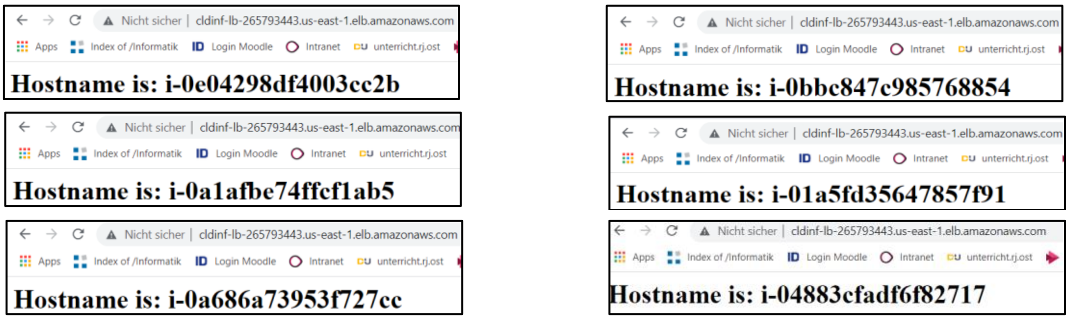


Figure 55-60 Hostnames shown on the website