# IS698 Cloud Computing using AWS

## PhotoSnap - Snapchat Clone

Under the guidance of Prof Samson Oni

Ankit Kumar Nath
LC46377

# Table Of Contents

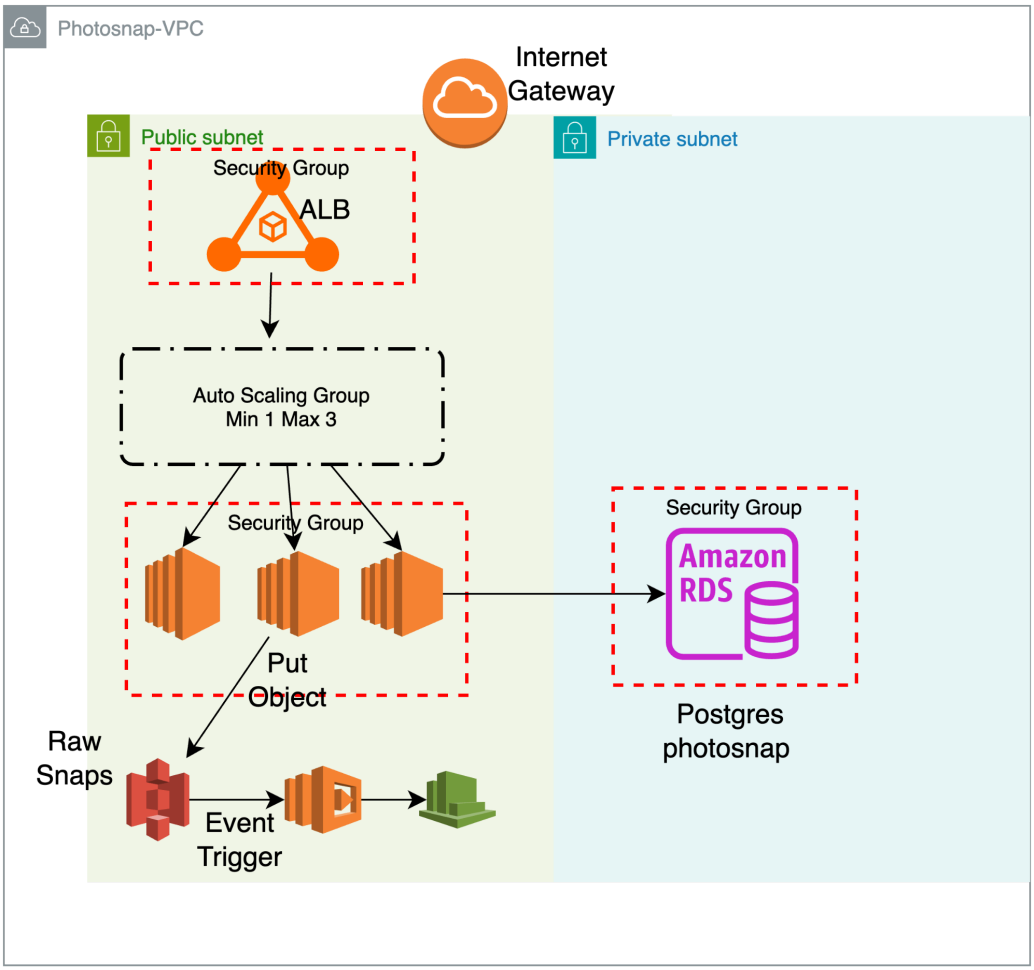
# 1. Introduction

**Photosnap** is a snapchat clone, where we have tried to replicate the core part of how snapchat works. A user would be able to login and signup into our app. The user will be able to choose an image and send it to other multiple users. The user can also view images that other users have sent. This was achieved through terraform, cloudformation and various **AWS** resources.

# 2. Methodology

The project is structured in such a way that we first setup the infrastructure through our terraform code, and then move onto the cloudformation for launching our **EC2** instances with the **Auto Scaling** feature. We also have a logging service which is achieved through **AWS Lambda** and can be viewed with help of **AWS Cloudwatch.** We will review the architecture followed by the implementation of the code.

# 3. Architecture

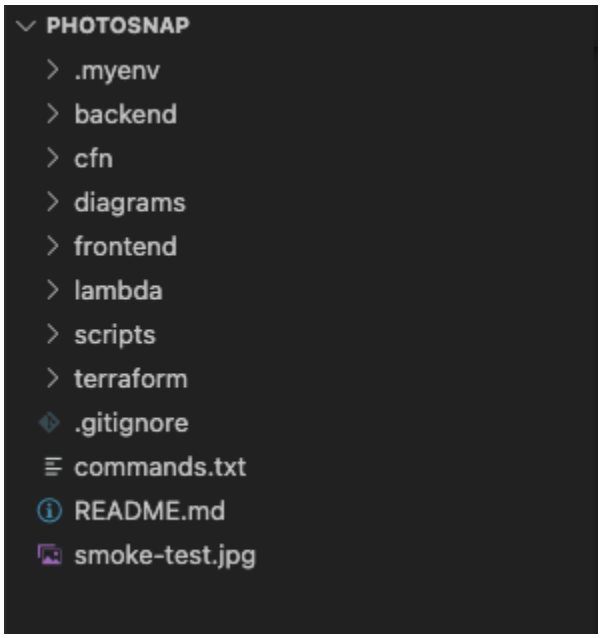


# 4. Code/Output

We start by setting up the project, we clone the PhotoSnap github repo. We can see the folder structure.

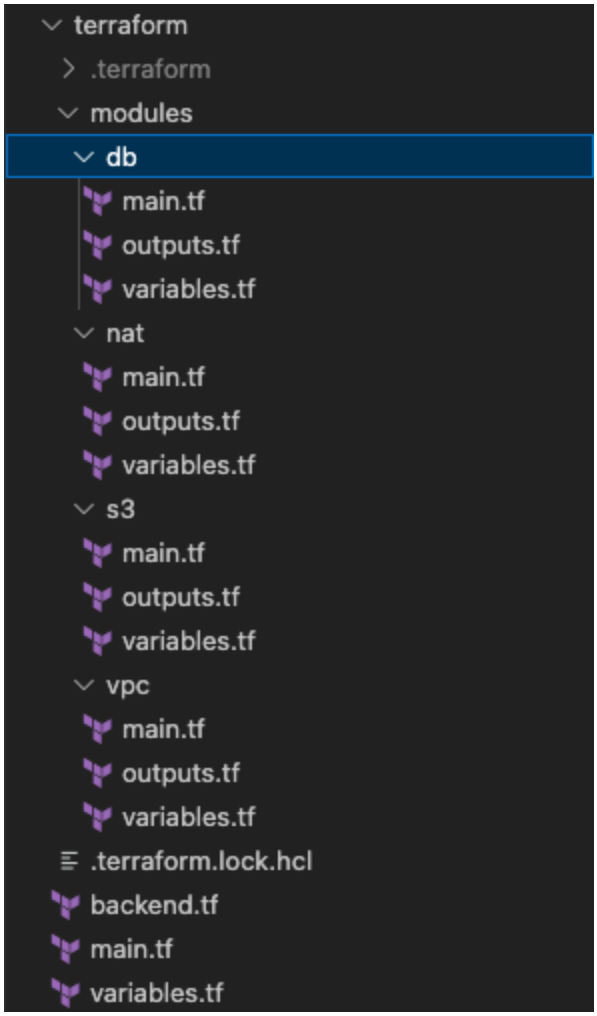
## 4.1 Terraform

As we can see the whole project is divided into sub folders, where we have separate folders for backend, cloudformation, frontend, lambda, scripts and terraform.




Let's start with setting up the Infrastructure for this project.

We start with terraform. We have a modules folder where we have the database, the s3 bucket (for storing the terraform state, the snap raw images, the lambda logger code), the vpc setup and the NAT gateway for instances hosted on private subnets.
We have two private and public subnets in our VPC.




Once we have everything setup, we run the below command. The **db_password** has the password for our **RDS** and the variable **enable_nat** enables or disables the **NAT** gateway depending if we want NAT or not based on the environment.

```shell
terraform apply \
  -var="db_password=password" \
  -var="enable_nat=false" \
  -auto-approve
```

```
module.db.aws_db_instance.snaps: Still creating... [5m0s elapsed]
module.db.aws_db_instance.snaps: Still creating... [5m10s elapsed]
module.db.aws_db_instance.snaps: Still creating... [5m20s elapsed]
module.db.aws_db_instance.snaps: Still creating... [5m30s elapsed]
module.db.aws_db_instance.snaps: Still creating... [5m40s elapsed]
module.db.aws_db_instance.snaps: Still creating... [5m50s elapsed]
module.db.aws_db_instance.snaps: Still creating... [6m0s elapsed]
module.db.aws_db_instance.snaps: Still creating... [6m10s elapsed]
module.db.aws_db_instance.snaps: Still creating... [6m20s elapsed]
module.db.aws_db_instance.snaps: Creation complete after 6m27s [id=db-XFCLKOOK7FQKWNWJXOV5XHCWCQ]

Apply complete! Resources: 16 added, 0 changed, 0 destroyed.

Outputs:

bucket_name = "photosnap-raw-snaps-4490772a"
db_endpoint = "photosnap-db.cktgkiokq2qf.us-east-1.rds.amazonaws.com"
private_subnet_ids = [
  "subnet-0ca19168146a01f82",
  "subnet-00b27aecdca21ea1a",
]
public_subnet_ids = "subnet-013441305b454278b,subnet-002a69f372656adc7"
vpc_id = "vpc-0ac3731a683126b80"
~/Desktop/MS/IS698/photosnap/terraform main*                                    7m 3s 15:24:19
❯
```

```
variables.tf  M
.gitignore
README.md

OUTLINE
TIMELINE

|DescribeDBInstances|
+-------------------+
|  available        |
|  14.12            |
|  db.t3.micro      |
+-------------------+
(END)

main*  ⊗ 0 ⚠ 0                                  Ln 35, Col 34  Spaces: 2  UTF-8  LF  Plain Text
```

## Your VPCs (2) Info

Last updated less than a minute ago | Actions | Create VPC

Find VPCs by attribute or tag

| Name | VPC ID | State | Block Public... | IPv4 CIDR | IPv6 CIDR | DHCP option set |
|---|---|---|---|---|---|---|
| – | vpc-05337bcae3a3cb2d4 | Available | Off | 172.31.0.0/16 | – | dopt-0b14cedacaf20b4da |
| photosnap-vpc | vpc-0ac3731a683126b80 | Available | Off | 10.0.0.0/16 | – | dopt-0b14cedacaf20b4da |

## Subnets (10) Info

Last updated less than a minute ago | Actions | Create subnet

Find subnets by attribute or tag

| Name | Subnet ID | State | VPC | Block Public... | IPv4 CIDR | IPv6 CIDR |
|---|---|---|---|---|---|---|
| private-10.0.4.0/24 | subnet-00b27aecdca21ea1a | Available | vpc-0ac3731a683126b80 \| pho... | Off | 10.0.4.0/24 | – |
| – | subnet-0b0fca268f3ca9e1b | Available | vpc-05337bcae3a3cb2d4 | Off | 172.31.32.0/20 | – |
| – | subnet-0eb432a1e91d22fd2 | Available | vpc-05337bcae3a3cb2d4 | Off | 172.31.0.0/20 | – |
| – | subnet-0916bfbd996342c33 | Available | vpc-05337bcae3a3cb2d4 | Off | 172.31.48.0/20 | – |
| – | subnet-088482eba63dac4e8 | Available | vpc-05337bcae3a3cb2d4 | Off | 172.31.80.0/20 | – |
| – | subnet-0fc250c1c259c9c7e | Available | vpc-05337bcae3a3cb2d4 | Off | 172.31.16.0/20 | – |
| public-10.0.1.0/24 | subnet-013441305b454278b | Available | vpc-0ac3731a683126b80 \| pho... | Off | 10.0.1.0/24 | – |
| private-10.0.3.0/24 | subnet-0ca19168146a01f82 | Available | vpc-0ac3731a683126b80 \| pho... | Off | 10.0.3.0/24 | – |
| – | subnet-0bc0c9faeeb0fe3c5 | Available | vpc-05337bcae3a3cb2d4 | Off | 172.31.64.0/20 | – |
| public-10.0.2.0/24 | subnet-002a69f372656adc7 | Available | vpc-0ac3731a683126b80 \| pho... | Off | 10.0.2.0/24 | – |

## Internet gateways (2) Info

Actions | Create internet gateway

Find internet gateways by attribute or tag

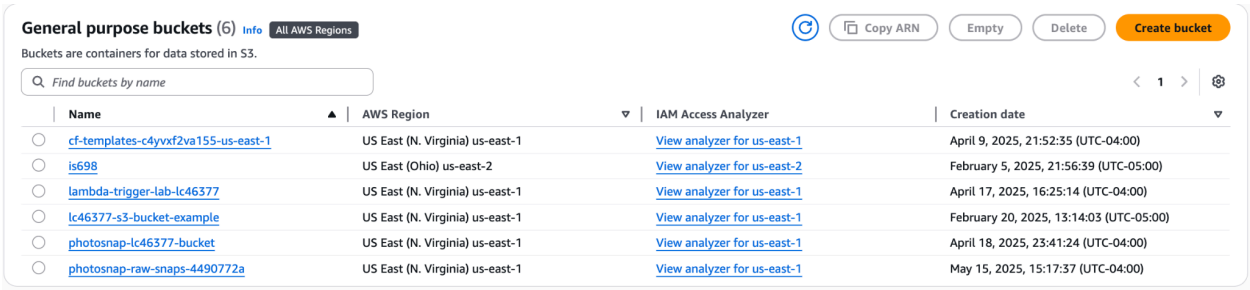
| Name | Internet gateway ID | State | VPC ID | Owner |
|---|---|---|---|---|
| photosnap-igw | igw-04d3dcfe80f64164f | Attached | vpc-0ac3731a683126b80 \| photosnap-... | 537124959881 |
| – | igw-0a0862bf7cb4c3f22 | Attached | vpc-05337bcae3a3cb2d4 | 537124959881 |

## Databases (1)

Group resources | Modify | Actions | Create database

Filter by databases

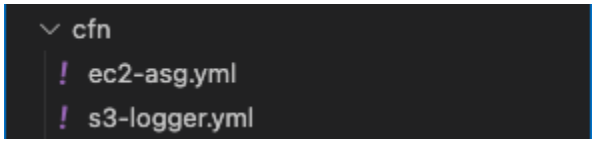
| DB identifier | Status | Role | Engine | Region ... | Size | Recommendations | CPU |
|---|---|---|---|---|---|---|---|
| photosnap-db | Available | Instance | PostgreSQL | us-east-1a | db.t3.micro | | 6.5 |




Now we can see that our resources are set up and our infrastructure is ready for the next steps.

## 4.2 CloudFormation

In this we handle the EC2 instances and the Auto Scaling Groups and also have a separate template for lambda to get logs when an image is uploaded. Let's discuss the folder structure. The ec2-asg.yml handles the setting up the instance and the user-data and ASG takes care of the auto scaling. The s3-logger.yml handles the logging when an image is uploaded.




In our ec2-asg.yml we can see that we have this user data.

Shell

```
 #!/bin/bash
set -e

# 1) Enable Python 3.8 & PostgreSQL 14 client
amazon-linux-extras enable python3.8 postgresql14
yum clean metadata -y

# 2) Install system packages
yum install -y git python3.8 postgresql

# 3) Create & activate a virtualenv
cd /home/ec2-user
python3.8 -m venv venv
source venv/bin/activate

# 4) Clone your backend & install Python deps
git clone https://github.com/lc46377/photosnap.git
cd photosnap/backend

# Upgrade pip, then install exactly your requirements
```

```
            pip install --upgrade pip
            pip install -r requirements.txt

            # 5) Bootstrap your database schema (snaps + auth + friendships)
            psql "host=${DbEndpoint} port=5432 user=${DbUsername}
password=${DbPassword} dbname=postgres sslmode=require" <<EOSQL
            CREATE EXTENSION IF NOT EXISTS "uuid-ossp";

            CREATE TABLE IF NOT EXISTS users (
              id            UUID PRIMARY KEY DEFAULT uuid_generate_v4(),
              username      TEXT UNIQUE NOT NULL,
              password_hash TEXT NOT NULL,
              created_at    TIMESTAMP WITH TIME ZONE DEFAULT now()
            );

            CREATE TABLE IF NOT EXISTS friend_requests (
              id          UUID PRIMARY KEY DEFAULT uuid_generate_v4(),
              from_user   UUID REFERENCES users(id) ON DELETE CASCADE,
              to_user     UUID REFERENCES users(id) ON DELETE CASCADE,
              status      TEXT     NOT NULL,  -- e.g.
"pending","accepted","rejected"
              created_at  TIMESTAMP WITH TIME ZONE DEFAULT now()
            );

            CREATE TABLE IF NOT EXISTS friendships (
              user_id    UUID REFERENCES users(id) ON DELETE CASCADE,
              friend_id  UUID REFERENCES users(id) ON DELETE CASCADE,
              created_at TIMESTAMP WITH TIME ZONE DEFAULT now(),
              PRIMARY KEY(user_id, friend_id)
            );

            CREATE TABLE IF NOT EXISTS snaps (
              id       UUID PRIMARY KEY DEFAULT uuid_generate_v4(),
              s3_key   TEXT NOT NULL,
              owner    UUID REFERENCES users(id) ON DELETE CASCADE,
              created_at TIMESTAMP WITH TIME ZONE DEFAULT now()
            );

            CREATE TABLE IF NOT EXISTS snap_recipients (
              snap_id    UUID REFERENCES snaps(id) ON DELETE CASCADE,
              user_id    UUID REFERENCES users(id) ON DELETE CASCADE,
              viewed     BOOLEAN DEFAULT false,
              created_at TIMESTAMPTZ DEFAULT now(),
              PRIMARY KEY(snap_id, user_id)
```

```
            );
            EOSQL

            # 6) Export runtime env-vars
            export SNAPS_BUCKET=${SnapsBucket}
            export DB_ENDPOINT=${DbEndpoint}
            export DB_USERNAME=${DbUsername}
            export DB_PASSWORD=${DbPassword}
            export AWS_REGION=${AwsRegion}

            # 7) Launch your Flask app under Gunicorn
            nohup /home/ec2-user/venv/bin/gunicorn app:app --bind 0.0.0.0:5000
&
```

So we install python 3.8 to avoid version conflicts, postgres, and the packages that we have in requirements.txt

```Shell
Flask==2.2.5
boto3>=1.26.0,<2.0.0
SQLAlchemy==2.0.40
psycopg2-binary==2.9.10
gunicorn==23.0.0
Flask-Cors==3.0.10
flask-jwt-extended==4.6.0
werkzeug==3.0.6
```

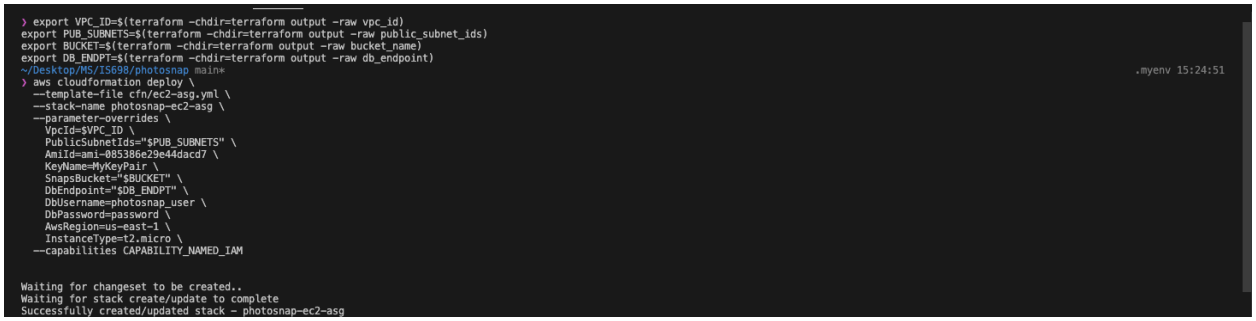
After installing these packages, we set up the tables for our data. So for all EC2 instances, we set up these packages so that our backend can communicate with the frontend.
Before we run our cloud formation template, we need to export the variables so that we can run our CloudFormation template

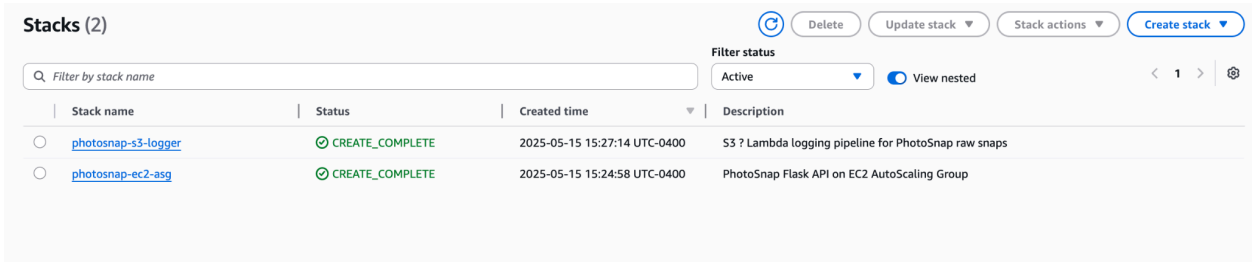
```Shell
export VPC_ID=$(terraform -chdir=terraform output -raw vpc_id)
export PUB_SUBNETS=$(terraform -chdir=terraform output -raw public_subnet_ids)
export BUCKET=$(terraform -chdir=terraform output -raw bucket_name)
export DB_ENDPT=$(terraform -chdir=terraform output -raw db_endpoint)
```

Run the cloud formation command for setting up the ec2-asg

```shell
aws cloudformation deploy \
  --template-file cfn/ec2-asg.yml \
  --stack-name photosnap-ec2-asg \
  --parameter-overrides \
    VpcId=$VPC_ID \
    PublicSubnetIds="$PUB_SUBNETS" \
    AmiId=ami-085386e29e44dacd7 \
    KeyName=MyKeyPair \
    SnapsBucket="$BUCKET" \
    DbEndpoint="$DB_ENDPT" \
    DbUsername=photosnap_user \
    DbPassword=password \
    AwsRegion=us-east-1 \
    InstanceType=t2.micro \
  --capabilities CAPABILITY_NAMED_IAM
```

Here we pass the public subnets, the database endpoints and the ec2 configurations so that all our scaled EC2 will follow the same configurations.



## 4.3 Logger

We create a zip of our logger code and send it to the s3 bucket using the command

```shell
RAW_BUCKET=$(terraform -chdir=terraform output -raw bucket_name)
aws s3 cp lambda/s3_logger.zip s3://$RAW_BUCKET/lambda/s3_logger.zip
```

Once we have the logger code in the s3 we can run our cloud formation code for the logger. We have separated our ec2 and logger because they are independent of each other keeping it modular and abstracted.

```shell
aws cloudformation deploy \
  --template-file cfn/s3-logger.yml \
  --stack-name photosnap-s3-logger \
  --parameter-overrides RawSnapsBucket=$RAW_BUCKET \
  --capabilities CAPABILITY_NAMED_IAM
```

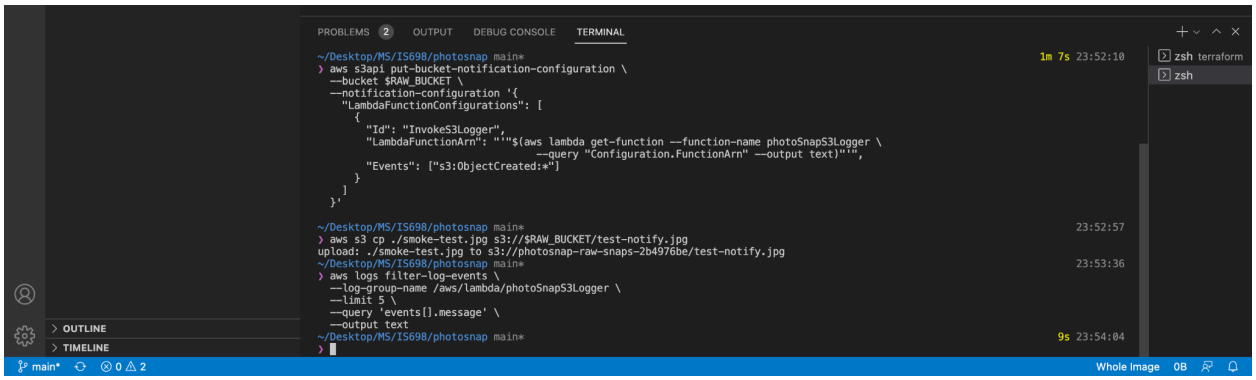
We have our architecture setup. Let's pull the ALB endpoint and set it in our frontend/.env file Our .env contains the url for which each api makes a request through.

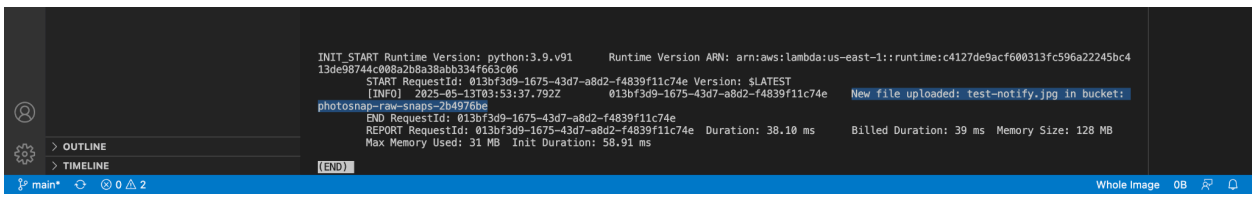
```shell
export ALB_URL="http://$(aws cloudformation describe-stacks \
  --stack-name photosnap-ec2-asg \
  --query "Stacks[0].Outputs[?OutputKey=='LoadBalancerDNS'].OutputValue" \
  --output text)"
echo "ALB → $ALB_URL"
```


Let's do a smoke test to test if our apis are working! Using the below command, we first make a request to our ALB endpoint which points to our EC2 where our backend is hosted. We also save our response and also try to get back our downloaded data.

```shell
curl -v -X POST $ALB_URL/upload \
  -H "Content-Type: application/json" \
  -d '{"filename":"smoke-test.jpg"}'
```

```
RESP=$(curl -s -X POST $ALB_URL/upload \
  -H "Content-Type: application/json" \
  -d '{"filename":"smoke-test.jpg"}')
PUT_URL=$(jq -r .put_url <<<"$RESP")
curl --upload-file ./smoke-test.jpg "$PUT_URL" && echo "uploaded"

GET_URL=$(jq -r .get_url <<<"$RESP")
curl -s "$GET_URL" -o downloaded.jpg && echo "downloaded"
```


## 4.4 EC2

Once we have everything setup, we need to check what's happening behind the scenes. We first login into our EC2 and actually see if the packages were installed properly along with the table creation.

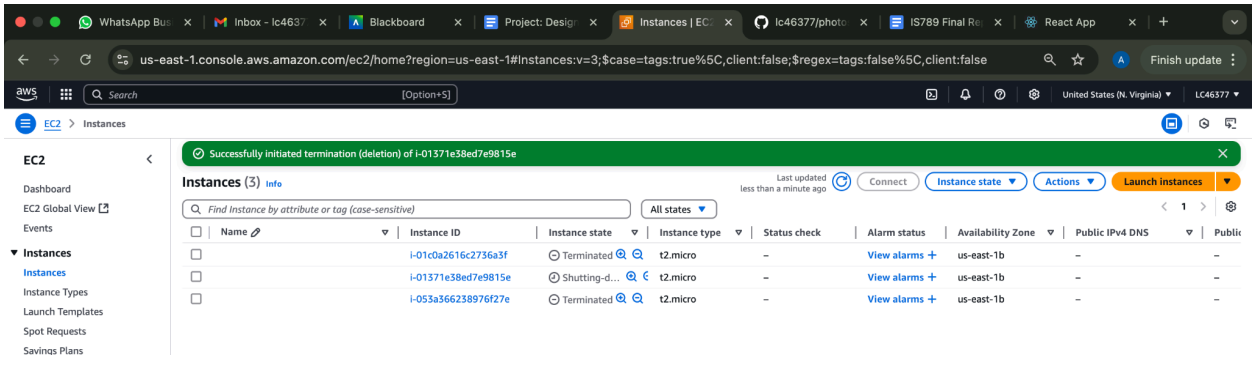
We need to add our IP so that we can SSH into the EC2, because we wouldn’t want to SSH from other IP’s

Shell

```
curl -s https://checkip.amazonaws.com
```

```
❯ curl -s https://checkip.amazonaws.com
73.191.110.121
~                                                    15:37:26
❯
```

```
/Desktop/MS/IS698                                    15:38:16
 ssh -i MyKeyPair.pem ec2-user@44.199.246.94
he authenticity of host '44.199.246.94 (44.199.246.94)' can't be established.
D25519 key fingerprint is SHA256:jBZXpMtHzXkpFcQx5uYtQ7eOCLMkYjm7QFRkYqtXsG8.
his key is not known by any other names.
re you sure you want to continue connecting (yes/no/[fingerprint])? yes
arning: Permanently added '44.199.246.94' (ED25519) to the list of known hosts.
  ,     #_
  ~\_  ####_        Amazon Linux 2
 ~~  \_#####\
 ~~     \###|       AL2 End of Life is 2026-06-30.
 ~~       \#/ ___
  ~~       V~' '->
   ~~~         /    A newer version of Amazon Linux is available!
     ~~._.   _/
        _/ _/       Amazon Linux 2023, GA and supported until 2028-03-15.
      _/m/'           https://aws.amazon.com/linux/amazon-linux-2023/

 package(s) needed for security, out of 7 available
un "sudo yum update" to apply all updates.
ec2-user@ip-10-0-1-95 ~]$
```

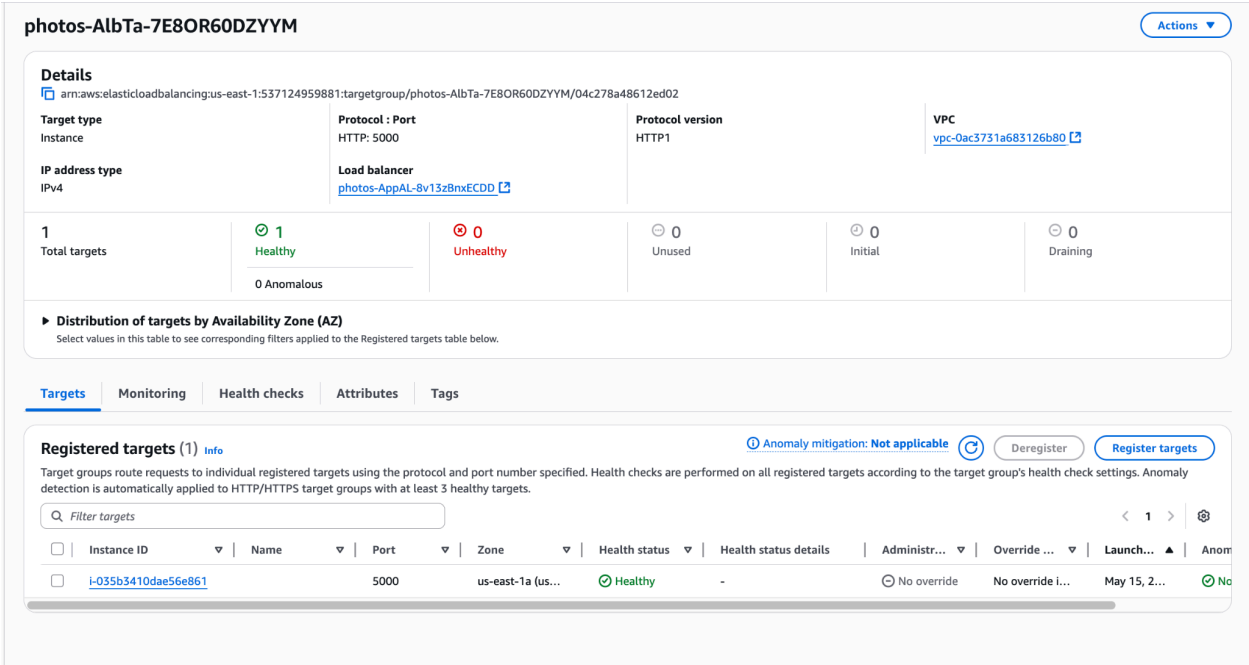



Shell

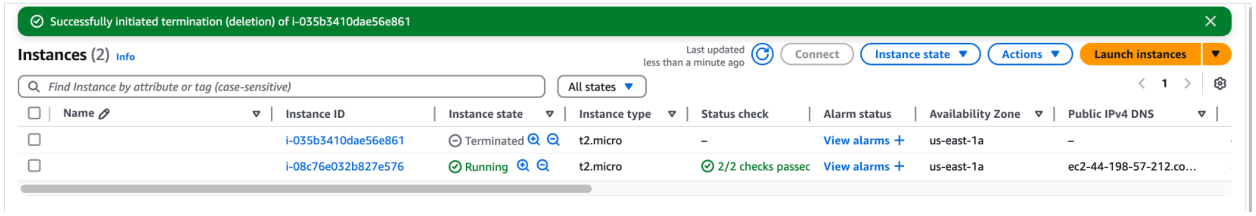
```
sudo tail -n 200 /var/log/cloud-init-output.log
```

```
IS698 — ec2-user@ip-10-0-1-95:~ — ssh -i MyKeyPair.pem ec2-user@44.1..
Downloading zipp-3.20.2-py3-none-any.whl (9.2 kB)
Installing collected packages: zipp, urllib3, typing-extensions, Six, PyJWT, ps
ycopg2-binary, packaging, MarkupSafe, jmespath, itsdangerous, greenlet, click,
werkzeug, SQLAlchemy, python-dateutil, Jinja2, importlib-metadata, gunicorn, Fl
ask, botocore, s3transfer, flask-jwt-extended, Flask-Cors, boto3
Successfully installed Flask-2.2.5 Flask-Cors-3.0.10 Jinja2-3.1.6 MarkupSafe-2.
1.5 PyJWT-2.9.0 SQLAlchemy-2.0.40 Six-1.17.0 boto3-1.37.38 botocore-1.37.38 cli
ck-8.1.8 flask-jwt-extended-4.6.0 greenlet-3.1.1 gunicorn-23.0.0 importlib-meta
data-8.5.0 itsdangerous-2.2.0 jmespath-1.0.1 packaging-25.0 psycopg2-binary-2.9
.10 python-dateutil-2.9.0.post0 s3transfer-0.11.5 typing-extensions-4.13.2 urll
ib3-1.26.20 werkzeug-3.0.6 zipp-3.20.2
CREATE EXTENSION
CREATE TABLE
CREATE TABLE
CREATE TABLE
CREATE TABLE
Cloud-init v. 19.3-46.amzn2.0.5 finished at Thu, 15 May 2025 19:26:46 +0000. Da
tasource DataSourceEc2.  Up 62.81 seconds
[2025-05-15 19:26:46 +0000] [3411] [INFO] Starting gunicorn 23.0.0
[2025-05-15 19:26:46 +0000] [3411] [INFO] Listening at: http://0.0.0.0:5000 (34
11)
[2025-05-15 19:26:46 +0000] [3411] [INFO] Using worker: sync
[2025-05-15 19:26:46 +0000] [3422] [INFO] Booting worker with pid: 3422
[ec2-user@ip-10-0-1-95 ~]$
```

We can also see the health status of our Target Group

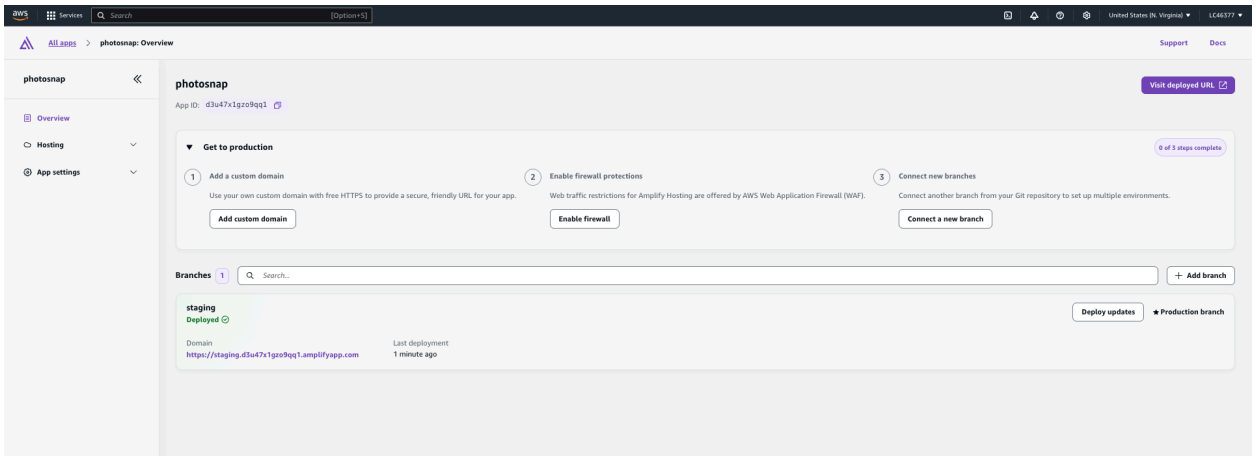



We can also terminate our instance and check if it triggers a new EC2

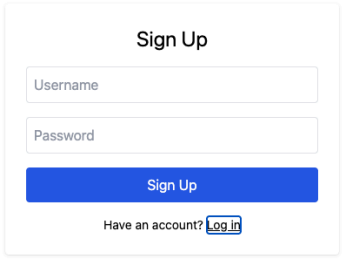



## 4.5 User Testing

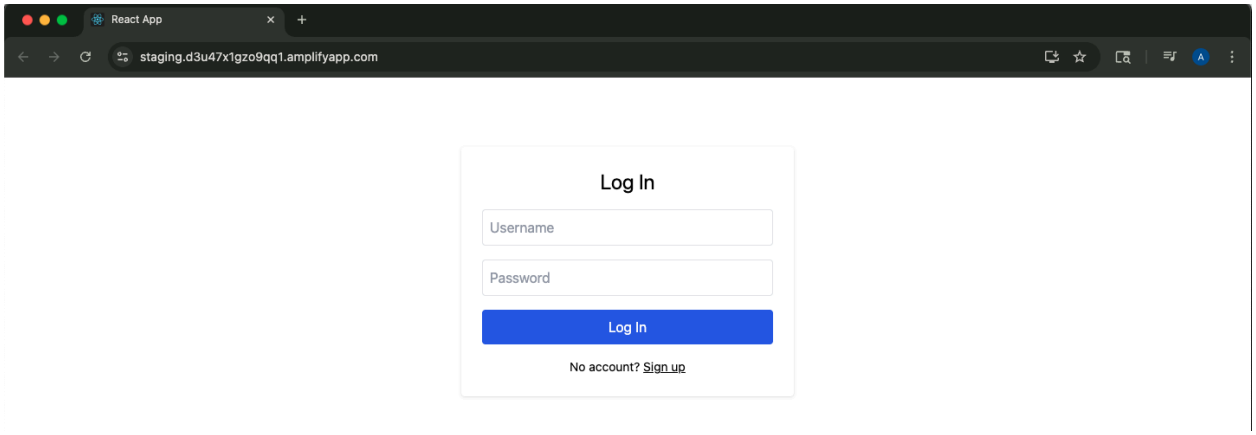
The user can login into our application that is hosted on the AWS server using AWS Amplify



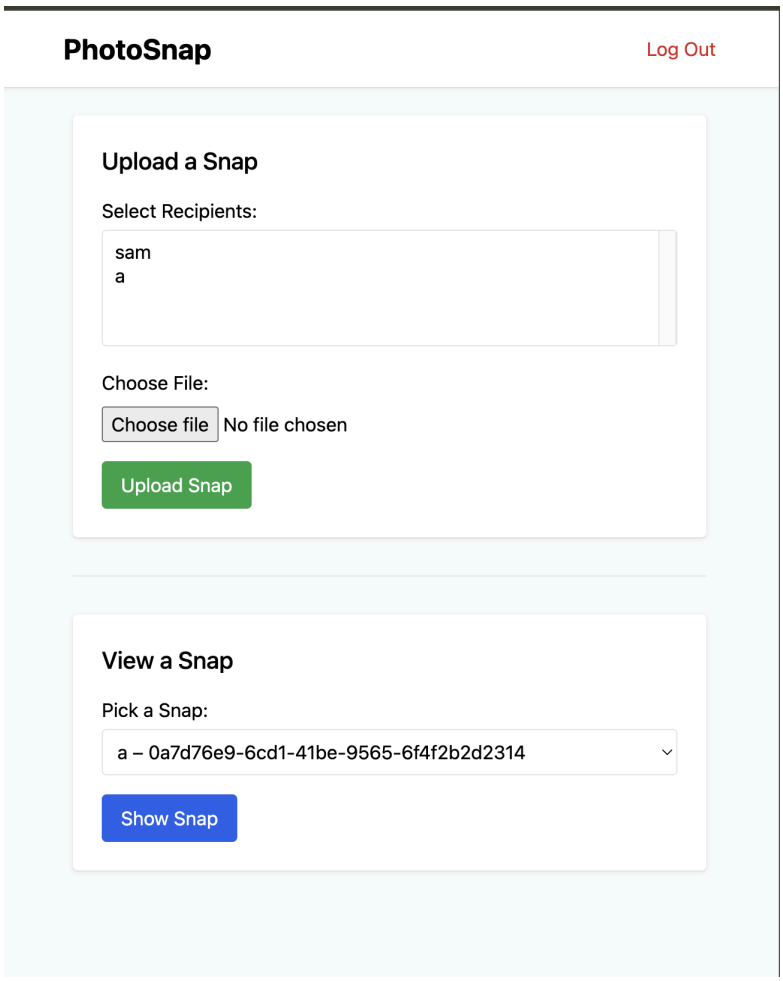

Create a account with us,

Login into the account



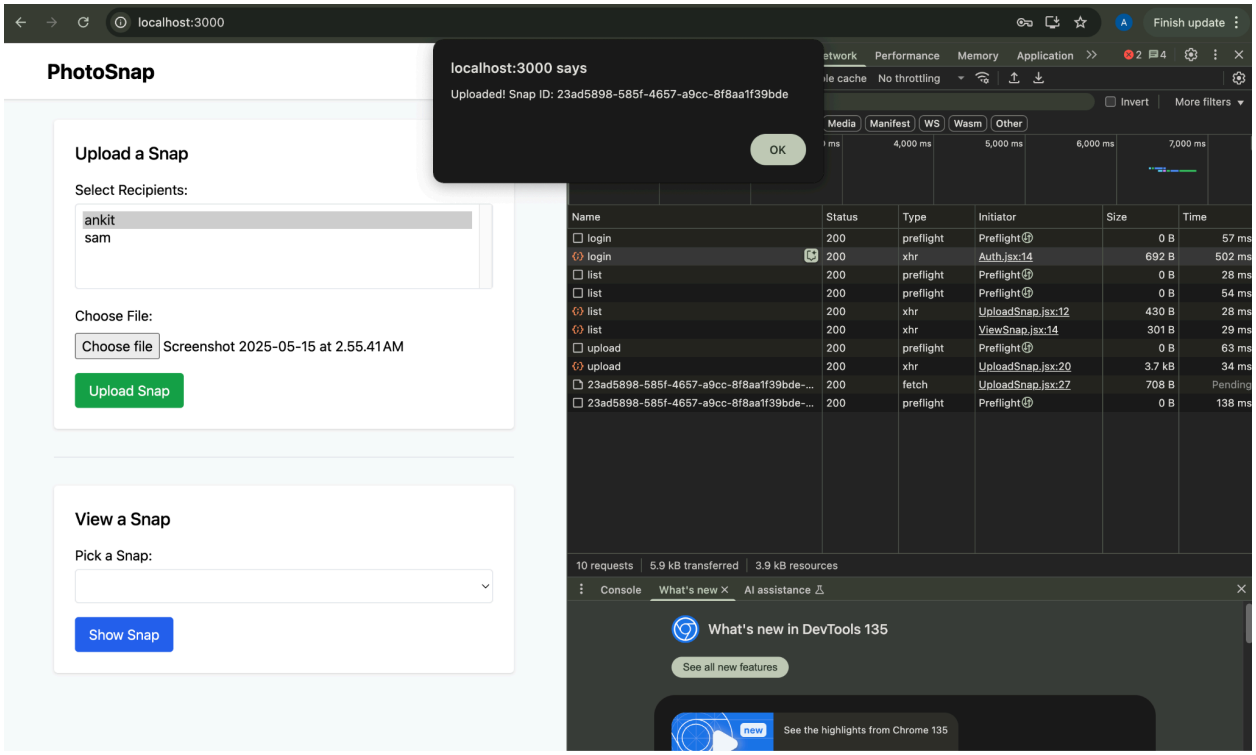

Choose from list of users and choose a file to send, also the user can view the snap that they might have received



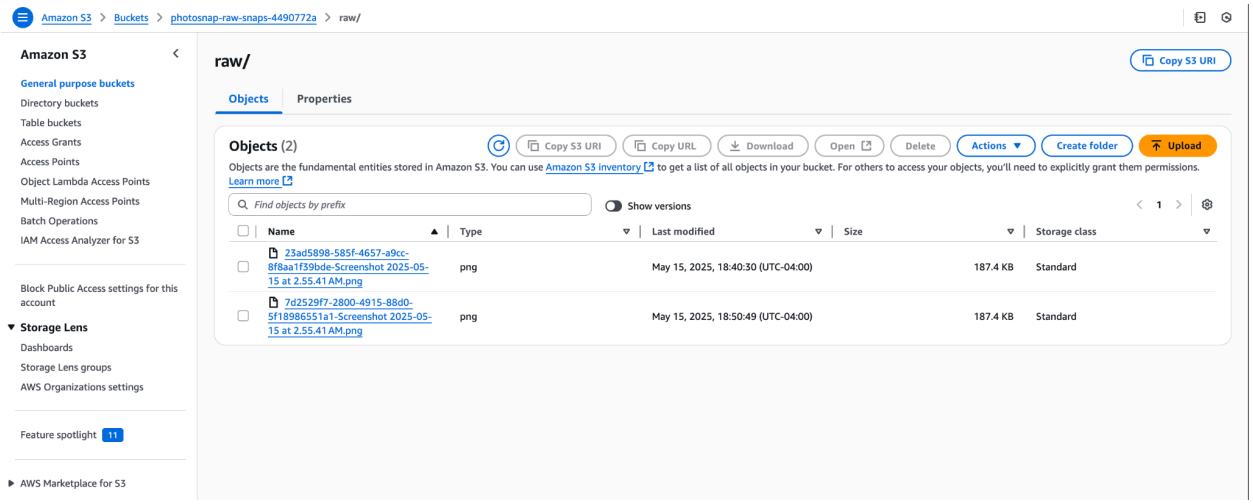
We can check the list of users present in our database through the EC2

```
[ec2-user@ip-10-0-1-107 ~]$ export PGSSLMODE=require
[ec2-user@ip-10-0-1-107 ~]$
[ec2-user@ip-10-0-1-107 ~]$ psql \
>   --host="$DB_ENDPOINT" \
>   --port=5432 \
>   --username="$DB_USERNAME" \
>   --dbname="$DB_NAME" \
>   -c "SELECT id, username FROM users;"
                  id                  | username
--------------------------------------+----------
 facaab8a-2ab7-48ef-b06d-bd8cc0aedaf0 | ankit
 6ae827b9-6776-41fc-a6a7-ae7e53427b29 | sam
(2 rows)
```

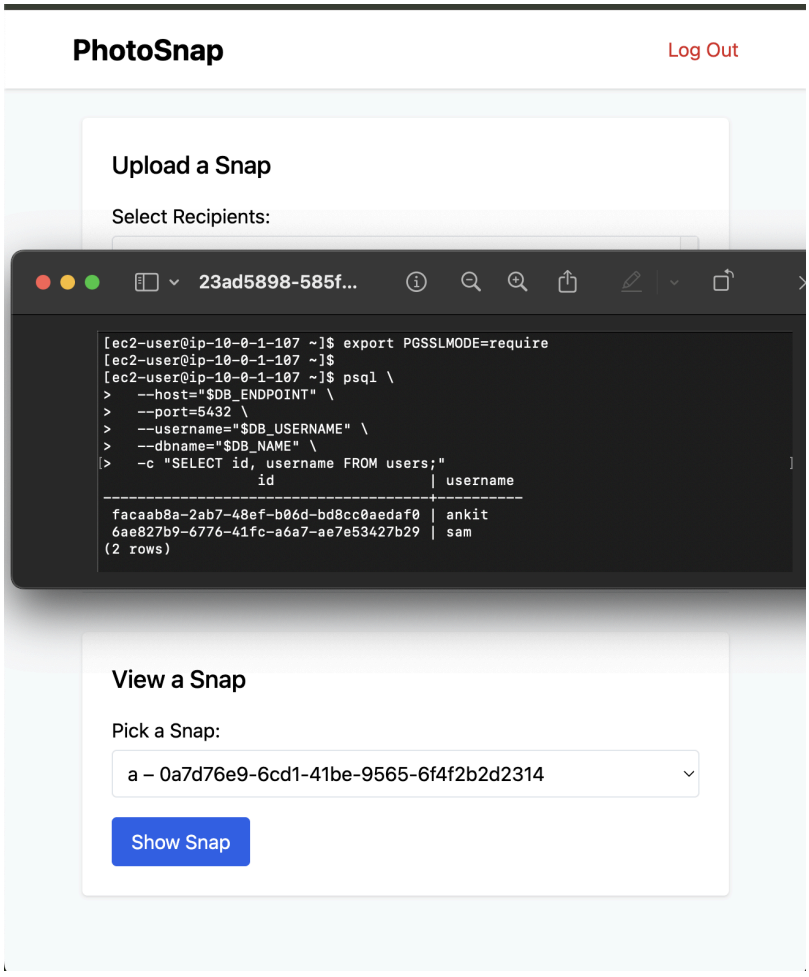
We can see that the api call was successful.




So once the user sends snap, it goes to our s3 bucket and gets stored as the raw file. We can also view our image from the S3 bucket.

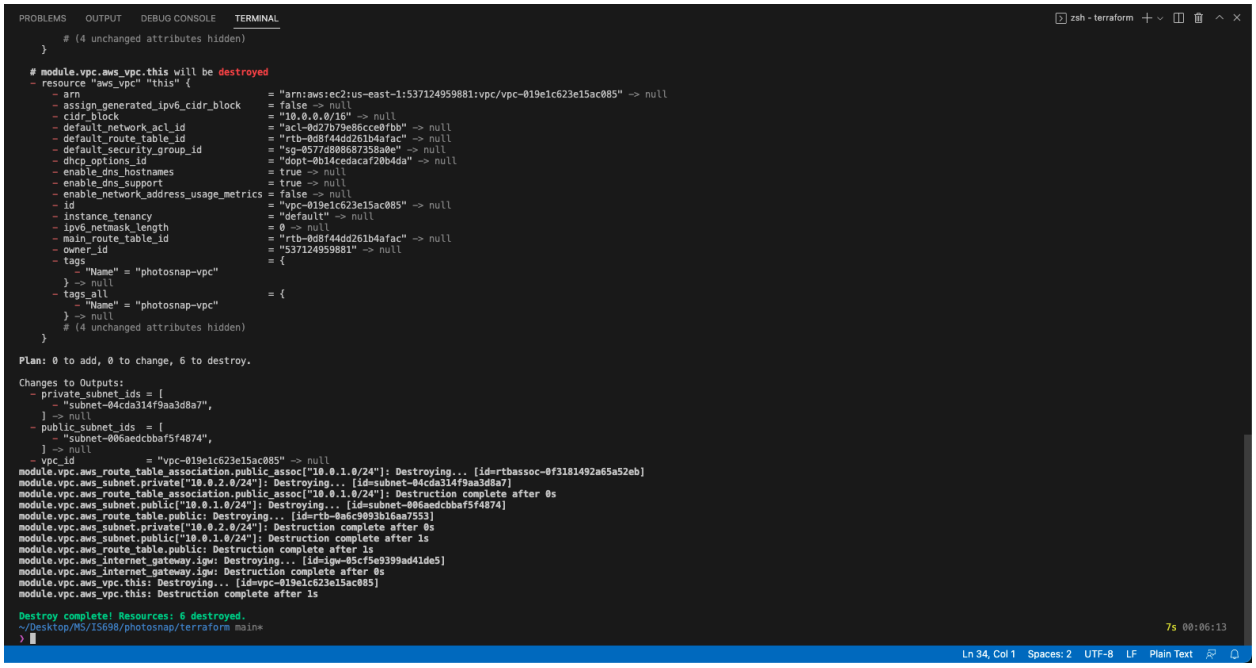



We can also view the snap sent by a different user




We have successfully developed a working snapchat clone that can send and receive images from other users.

We can destroy our AWS Infrastructure to avoid incurring cost 😂

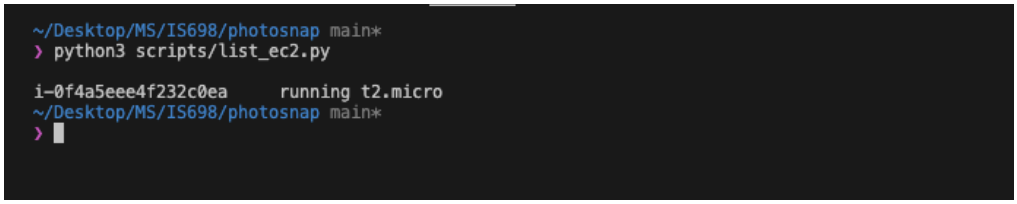



We could also run certain scripts to get some insights if we face errors. The scripts are located in the script folder.

This is the list_ec2.py that lists current ec2 instances for us.

Python

```python
import boto3

def main():
    ec2 = boto3.client("ec2")
    resp = ec2.describe_instances(
        Filters=[{"Name": "instance-state-name", "Values": ["running"]}]
    )
    for reservation in resp.get("Reservations", []):
        for inst in reservation.get("Instances", []):
            iid = inst.get("InstanceId")
            state = inst.get("State", {}).get("Name")
            typ = inst.get("InstanceType")
            print(f"{iid}\t{state}\t{typ}")

if __name__ == "__main__":
    main()
```

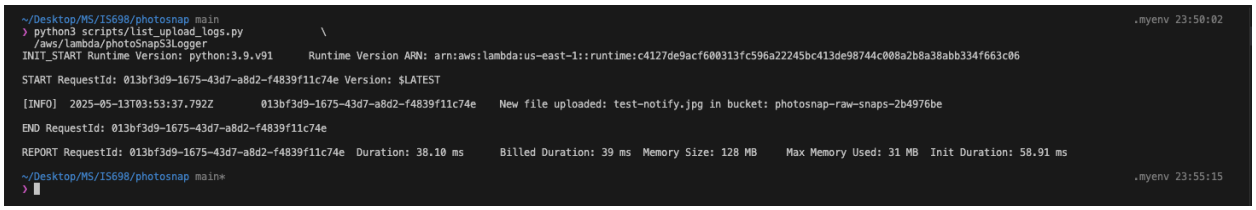



list_upload_logs.py for fetching the logs from our lambda function.

Python

```python
import boto3
import sys

def main():
    log_group = sys.argv[1] if len(
        sys.argv) > 1 else "/aws/lambda/photoSnapS3Logger"

    logs = boto3.client("logs", region_name="us-east-1")

    resp = logs.describe_log_streams(
        logGroupName=log_group,
        orderBy="LastEventTime",
        descending=True,
        limit=1
    )
    streams = resp.get("logStreams", [])
    if not streams:
        print(f"No log streams found in {log_group!r}")
        return

    stream_name = streams[0]["logStreamName"]

    # Fetch up to 10 of the latest events
    events = logs.get_log_events(
        logGroupName=log_group,
        logStreamName=stream_name,
        limit=10,
        startFromHead=False
    )["events"]

    if not events:
        print("No log events found.")
        return

    # Print each message
    for evt in events:
```

```
        print(evt["message"])


if __name__ == "__main__":
    main()
```




## 5. Challenges faced and solutions implemented

Many challenges were faced during project implementation.

1. ALB was not set up properly because it needs at least 2 public subnets to be set up properly. Solution was to implement 2 public and 2 private subnets so that the ALB can be launched without any issues.
2. Postgres specific version needs t2.micro instead of t3.micro storage, resolved by switching from t3 to t2.micro.
3. SSH into the EC2 launched from the cloudformation template from my IP, resolved by adding my ip into the security group
4. The target groups were not healthy, resolved by checking the logs of the EC2.
5. Packages were not installed due to version mismatch of python, resolved by finding the right versions for the python 3.8
6. Insertion of data into table due to wrong schema mismatch, fixed the schema
7. CORS policy related errors, resolved by adding OPTIONS in flask app.

These challenges were resolved after carefully analyzing the logs, reading articles and with help of resources online.

## 6. Future Works

We have set up the infrastructure, and have connected the frontend and backend. Frontend needs proper design and a fix needed to CORS policy. Backend needs to handle more edge cases. Scalability test is also required for photosnap.

Frameworks & Tools: React for Frontend, Flask for Backend, Draw.io for Architecture, Git for version control.