Index

1. Introduction

This is just a tutorial of how we built a webform in .NET that talks through IAM/S3 to AWS DynamoDB and is using a Docker Swarm setup on a scalable AWS EC2-instance setup. I wrote this guide for my own good (learning by describing), but sharing it in case someone else might benefit from it too.

Link to repository: https://github.com/mymh13/swarm-dotnet-test

1.1 Pre-requisites

AWS account, AWS CLI installed, SSH key, AWS key pair, and basic cloud and .NET knowledge

2. Infrastructure basics

In this phase we will do the rudimentary basics: Create a security group for the swarm and launch EC2 instances. Further infrastructure (database) will be added later.

2.1 Create a Security Group

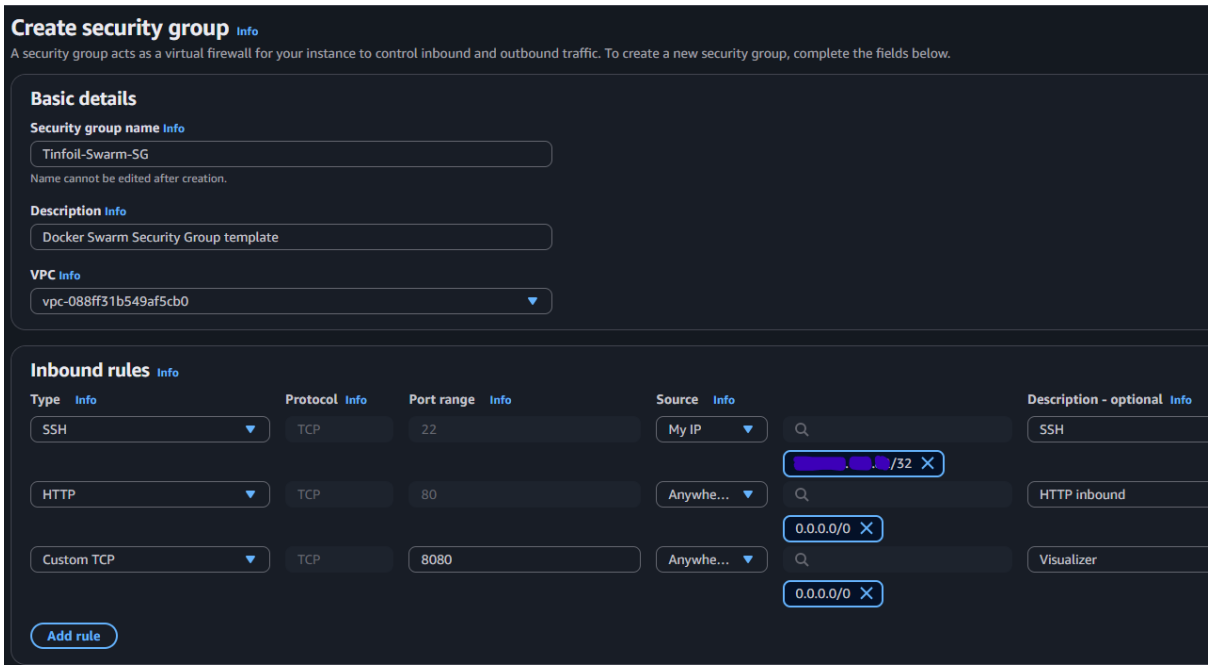
The SG sets the rules and boundaries for the system that uses that particular SG. In this first step, we want to create a SG that primarily handles the inbound rules for our Swarm network.

2.1.1 Create the initial inbound rules

In the AWS console, navigate to EC2 > Security Groups > Create Security Group. Fill in the following:

Name: Chose something descriptive so you know what role the SG plays. You will end up with many SGs eventually and then it will be hard to browse through them.
Description: Something descriptive, see above
VPC: use default VPC for this template, if you can do custom then you do not need this guide 😊
Inbound rules:

Type	Protocol	Port	Source	IP	Description
SSH	TCP	22	My IP	[yourIPhere]	SSH
HTTP	TCP	80	IPv4	0.0.0.0/0	HTTP inbound
Custom TCP	TCP	8080	IPv4	0.0.0.0/0	Visualizer

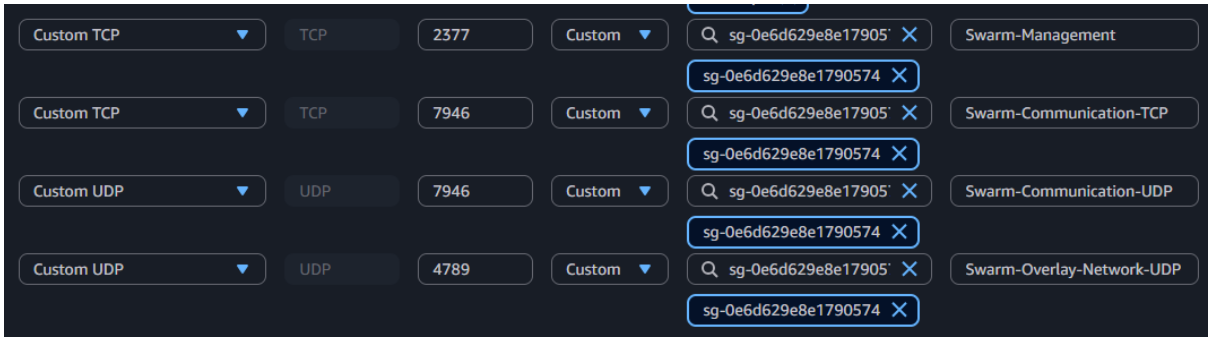


At this point you want to create the security group, there will be four more inbound rules but they will use self-reference so you want to create the SG before it can refer to itself!

2.1.2 Update the Security Group

We need to add the final inbound rules. Edit the Security Group. In the IP/search window, click the window and choice your SG you just created to self-reference.

Do note UDP on the two last ones! It should look something like this:



2.1.3 What the hell did we just do

Before we move on, let us briefly explain what this does:

- SSH (TCP 22) – Lets you connect to the server via SSH. We lock it down to your own IP only, so no one else can try to log in.
- HTTP (TCP 80) – This allows anyone on the internet to reach our application in the browser.
- Visualizer (TCP 8080) – Docker Swarms visualizer tool runs on port 8080. This rule makes us able to reach and view the visualizer in our browser.
- Swarm Management (TCP 2377) – Used for communication between the swarm manager and the worker nodes. Self-referenced so only other servers in the same SG are allowed in.
- Swarm Communication (TCP 7946 + UDP 7946) – These ports handle internal gossip (yes, that is the actual term in distributed systems! It means each node spread information to its neighbours) and node discovery within the system. Self-reference = self-contained, only swarm members allowed.
- Overlay Network (UDP 4789) – This port carries actual container traffic between swarm nodes over the same overlay network. Self-referenced, internal traffic inside the swarm.

2.1.4 Important note regarding SSH inbound:

Personally, I am always on VPN and my IP changes frequently. This means I have to consider how to handle my SSH IP inbound. In a brief project like this I chose to edit the SSH rule and adjust to reflect my new IP now and then, but for a longer term – and especially if you are more than one person handling this node – you need a more robust solution.

2.1.5 Standard vs custom ports

- Standard: TCP 22 is standard SSH-port. TCP 80 is standard HTTP- (web) traffic.
- Widely used: TCP 8080 is not an official standard, but a widely used convention for dashboards and dev tools.
- Custom: TCP 2377, TCP 7946 + UDP 7946 and UDP 4789 are specific to Docker Swarm.

Custom ports are chosen by the system- or application designer, it is important to know the standards and commonly used ones, so we don't accidentally re-use them for custom roles.

2.1.6 TCP vs UDP protocols

Most internet traffic uses one of the two protocols: TCP or UDP.

- TCP (Transmission Control Protocol) – It sets up a connection, checks that messages arrive, re-sends everything that gets lost. It is reliable but has a bit more overhead. We use this for SSH and HTTP because there accuracy is more important than speed.
- UDP (User Datagram Protocol) – It is sent without checking if it arrives, thus making it much faster and uses less overhead than TCP, but delivery is not guaranteed! This is useful for Docker Swarm's internal networking, especially if we are working idempotent as we can accept lost packets.

In short: TCP for reliable traffic, UDP for fast cluster chatter and data transport.

2.1.7 SG-Summary

We have create a Security Group that will handle inbound rules, boundaries and internal traffic.

2.2 Launch EC2 Instances

EC2 Instances is basically just Virtual Machines (VM) but let us stick to AWS-terminology. They will work as "servers" (nodes/hosts might be more appropriate description) for our Swarm. At this point we could also have considered other VMs for roles like a Bastion Host, or a Database or something else, but let us keep this tutorial simple for now and just focus on the Docker Swarm.

2.2.1 First let us create the instances

AWS Console > EC2 > Instances > Launch Instance

Name: Just like with the SG group and classes/methods, always use descriptive names!
AMI: Amazon Linux 2023
Instance type: t3.small
Key pair: This was a pre-req 😊 chose your already pre-set-AWS-key-pair for this
Network settings:

- VPC – default VPC
- Subnet – default Subnet
- Auto-assign public IP – enable
- Firewall / Security Group – Select existing (the one you just created in step 1)

Advanced details > User Data: add a start script for the instance:

```
#!/bin/bash
dnf update -y
dnf install -y docker
systemctl enable --now docker
usermod -aG docker ec2-user
```

2.2.1.1 ..and this script does what?

`dnf update -y` - Updates all existing software on the instance. The -y flag means "say yes to all prompts" so the process doesn't stop to ask

`dnf install -y installs docker` - Installs Docker from the Amazon Linux package repository. Again, -y auto-confirms

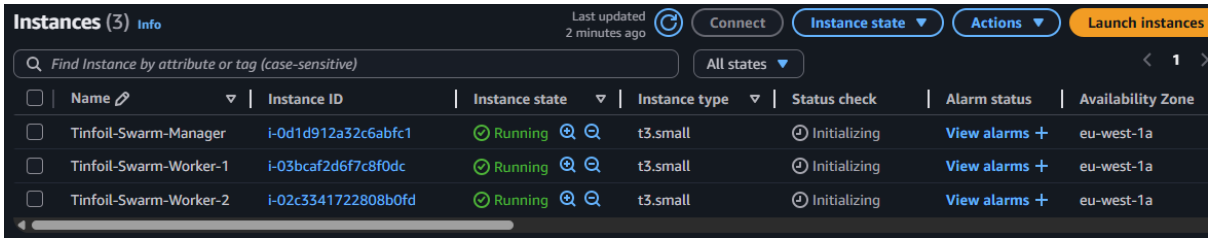
`systemctl enable –now docker` - Tells the system to start Docker right away (--now) and to also start it automatically on every reboot (enable)

`usermod -aG docker ec2-user` - Adds the default AWS login user (ec2-user) to the docker group, so you can run Docker commands without needing sudo every time

2.2.2 Summary, launch and multiples

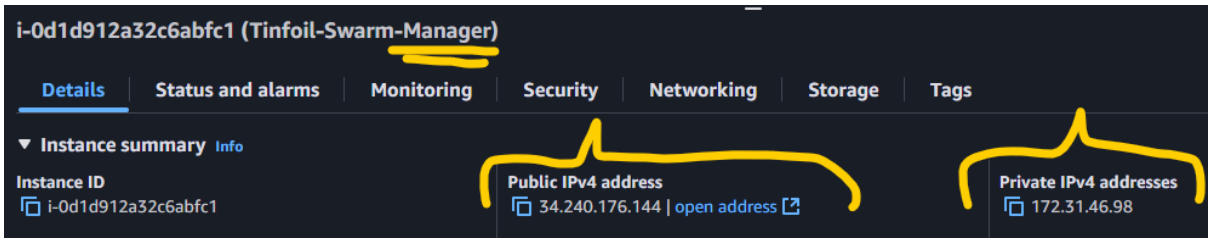
On your right hand, you should have a “Summary” window, with a box named “number of instances” – chose 3 and then click Launch instance.

Then go your EC2 > Instances and rename the second and third Manager-instances to Worker-1 + 2:



2.2.3 Note Instance IPs

View the instances (in the version above, September 2025, you can just click the instance tick box to the left and you get a quick view of the instance in a window below the instances). You want to make sure you have the Public IP and the Private IP for the three instances. I copy-pasted them down in a markdown document for easy access.



2.2.4 EC2-Summary

We have set up three EC2 Instances, one working as a Master and two Workers nodes.

2.3 Initialize Docker Swarm via SSH

I hope you have got the IPs handy and SSH + Key Pair is set up. In this phase we will initialize Docker Swarm on our EC2 Instances and verify the cluster. Let’s go.

2.3.1 Create a nifty helper-document (optional but recommended)

I used Obsidian and wrote in markdown, but anything will do. What I did was having a number of sections where I listed the public/private IPs of the Instances and a copy-paste command ready to SSH into the Instances. It should look something like this:

```
Swarm Manger
```

```
- public IP - 85.482.592.492
- private IP - 172.31.46.98
ssh -i ~/.ssh/<your-key-name>.pem ec2-user@85.482.592.492
```

Do note:

- This assumes your SSH key is in the default user/.ssh/ directory. If you are a Windows user I can highly recommend to always SSH keys in the C:\Users\<username>\.ssh directory
- The public IP in the example above has randomized numbers, this is information you should protect so make sure you do not save this information open in – for example – your project folder (if you use that for a GitHub repo or similar)

2.3.2 Connect to the Manger and initialize Swarm

Use the SSH command found in 2.3.1 to SSH into your Manager node:

```
ssh -i ~/.ssh/your-key.pem ec2-user@<manager-public-ip>
```

Then we will initialize the Swarm on the Manager:

```
sudo docker swarm init --advertise-addr <manager-private-ip>
```

We should get a verification that looks similar to this:

```
Swarm initialized: current node (xyz123) is now a manager.

To add a worker to this swarm, run the following command:

    docker swarm join --token SWMTKN-1-xxx... <manager-private-ip>:2377
```

Copy the join-command and save it to your helper-document.

```
sudo docker swarm join --token SWMTKN-1-<long-string> <manager-private-
ip>:2377
```

2.3.2.1 I got most of that. I think

No worries, let me elaborate what you just did.

- By SSHing into the Manager node and running docker swarm init, we turned that server into the “brain” of the swarm.
- The `--advertise-addr <manager-private-ip>` flag makes sure other servers connect over the private AWS network (faster and more secure than public IP).
- Docker then gave us a join token and command. This is like a one-time password that worker nodes will use to join the swarm. Copy it somewhere safe — you’ll need it in the next step.

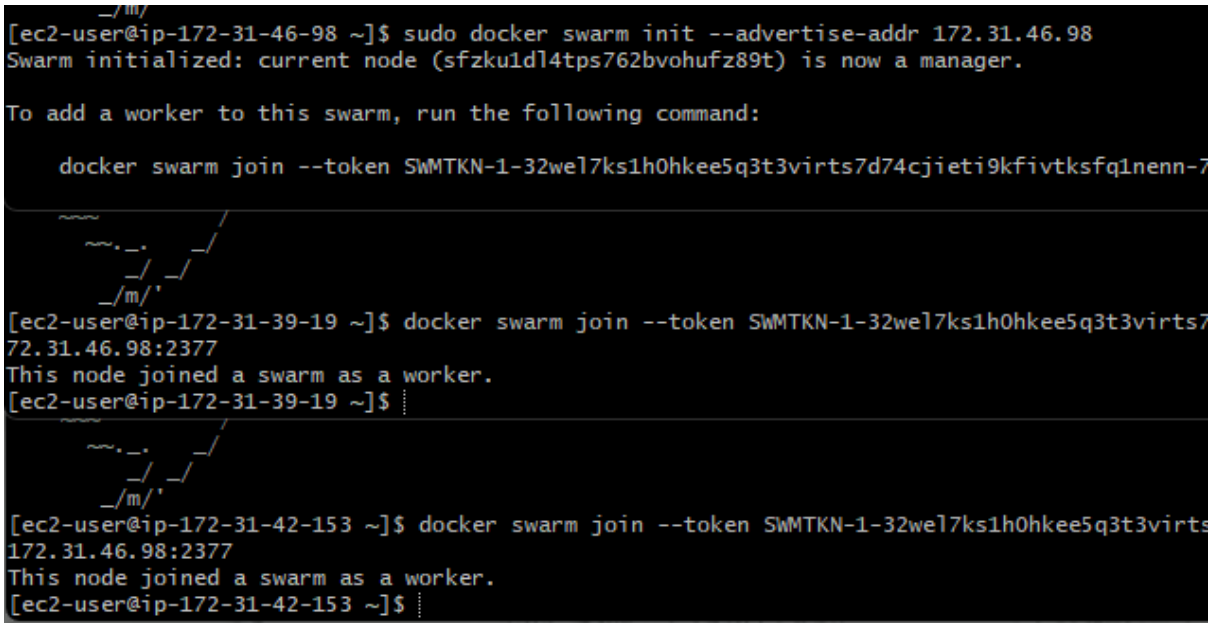
2.3.3 Let us add the Worker nodes

Basically, we will join the existing Swarm as worker nodes, there is not much else to it. Open a new terminal window and run these commands:

```
ssh -i ~/.ssh/your-key.pem ec2-user@<worker1-public-ip>
# Run the join command from step 2.3.2
sudo docker swarm join --token SWMTKN-1-xxx... <manager-private-ip>:2377
```

Repeat the above step but replace worker1-public-ip with worker2's public IP. I do not think we need to explain the command in-depth, as you see we use "swarm join" which should be self-explanatory.

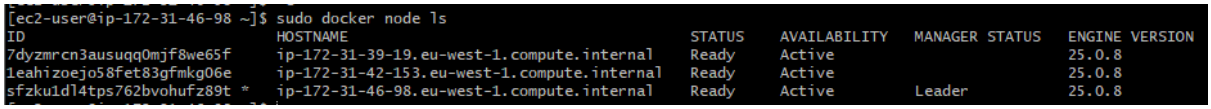
You should have three terminal windows that look something like this:



2.3.3.1 Let us quickly verify the Swarm cluster

In the Manager nodes' terminal, type this: `sudo docker node ls`

You should see this:



2.3.4 Swarm-Summary

We have initialized a Docker Swarm on our Manager, joined with Workers and verified it.

2.4 Deploying the services manually

Ideally this is something that we have an automatic solution for, some IaC solution would be preferable. But this is just a basic template to grasp the concept so we will deploy manually.

2.4.1 Create a Docker Compose file

On the Manager node, we will create the stack file by typing (pasting) this:

```
cat > docker-stack.yml << 'EOF'
version: "3.8"
services:
  web:
    image: nginx:stable-alpine
    deploy:
```

```
      replicas: 3
      restart_policy:
        condition: on-failure
      update_config:
        parallelism: 1
        delay: 5s
    ports:
      - "80:80"
    networks: [webnet]

  viz:
    image: dockersamples/visualizer:stable
    deploy:
      placement:
        constraints: [node.role == manager]
    ports:
      - "8080:8080"
    volumes:
      - /var/run/docker.sock:/var/run/docker.sock
    networks: [webnet]

networks:
  webnet:
    driver: overlay
EOF
```

2.4.1.1 Let us not go into all details, but roughly what this does is this

The cat > docker-stack.yml << 'EOF' ... EOF command is a way to create a new file (docker-stack.yml) and paste content directly into it. Everything between the two EOF markers becomes the file content.

This file is written in YAML, which is very sensitive to spacing and indentation — so it's best to copy-paste rather than type it manually.

The stack defines two services:

- Web: An nginx server with 3 replicas, exposed on port 80
- Viz: A Docker Swarm visualizer, only running on the manager node, exposed on port 8080 (we set that port in the SG but the Manager runs the Viz itself)

At the bottom, it also defines a custom overlay network (webnet) that lets the services talk to each other across swarm nodes.

2.4.2 Deploy the freshly created Stack

Still on the Manager node, run the deploy command:

```
sudo docker stack deploy -c docker-stack.yml myappname
```

```
[ec2-user@ip-172-31-46-98 ~]$ sudo docker stack deploy -c docker-stack.yml myappname
Creating network myappname_webnet
Creating service myappname_viz
Creating service myappname_web
```

This tells Docker Swarm to deploy the stack file (-c docker-stack.yml) and give the stack the name myappname.

2.4.3 Verify the deployment

```
# Show all stacks currently running in the Swarm
sudo docker stack ls

# List the services inside the stack, along with how many replicas are running. If we followed the guide you should see web with 3/3 replicas and viz with 1/1 – since we have 3x instances but only the Manager runs the viz.
sudo docker service ls

# Detailed info about each service: which node it is running on, current state, specific container ID. This is useful when troubleshooting.
sudo docker service ps myappname_web
sudo docker service ps myappname_viz
```

The comment above each command explains them further in-depth.

```
[ec2-user@ip-172-31-46-98 ~]$ sudo docker stack ls
NAME         SERVICES
myappname    2
[ec2-user@ip-172-31-46-98 ~]$ sudo docker service ls
ID             NAME             MODE         REPLICAS   IMAGE                             PORTS
orr1o0zokn16   myappname_viz    replicated   1/1        dockersamples/visualizer:stable   *:8080->8080/tcp
oq6uit43aetn   myappname_web    replicated   3/3        nginx:stable-alpine               *:80->80/tcp
```

```
[ec2-user@ip-172-31-46-98 ~]$ sudo docker service ps myappname_web
ID             NAME              IMAGE                NODE                                          DESIRED STATE   CURRENT STATE            ERROR     PORTS
pvilkvk75478   myappname_web.1   nginx:stable-alpine   ip-172-31-46-98.eu-west-1.compute.internal    Running         Running 3 minutes ago
z78t0hc6xyfj   myappname_web.2   nginx:stable-alpine   ip-172-31-42-153.eu-west-1.compute.internal   Running         Running 3 minutes ago
eg510zadh0b3   myappname_web.3   nginx:stable-alpine   ip-172-31-39-19.eu-west-1.compute.internal    Running         Running 3 minutes ago
[ec2-user@ip-172-31-46-98 ~]$ sudo docker service ps myappname_viz
ID             NAME              IMAGE                             NODE                                          DESIRED STATE   CURRENT STATE            ERROR
    PORTS
nzat3lxmepib   myappname_viz.1   dockersamples/visualizer:stable   ip-172-31-46-98.eu-west-1.compute.internal    Running         Running 3 minutes ago
```

2.4.4 Deploy-Summary

We have created and deployed the Docker Compose-file on the Manager and verified it.

2.5 Test web service and see if the service can scale

Now let's test that the web service is accessible, and then try scaling it up and down.

2.5.1 Let us first verify web access

Open a browser and go to the public IP of any node:

`http://<public-ip-of-any-node>`

You should see the default Nginx welcome page. Then check the visualizer service:

`http://<manager-public-ip>:8080/`

If everything is configured correctly, you'll see a live visualization of your swarm.

- Port 80 needs to be open for web access.
- Port 8080 needs to be open for the visualizer.

If either page doesn't load, double-check your Security Group and your stack file to make sure the correct ports are open to the right IPs.

2.5.2 Scale the services

Keep that Visualizer browser tab open, and run these commands on the Manager node:

```
# Scale up nginx to 5 replicas
sudo docker service scale myappname_web=5

# Check scaling
sudo docker service ps myappname_web

# Scale back down to 3
sudo docker service scale myappname_web=3
```

You should see containers appear and disappear in the visualizer as the service scales up and down.

Do note: Scaling here means adding or removing containers, not EC2 instances. Your swarm is still running on the same set of servers. You're just changing how many copies of the nginx container run across them.

2.5.3 Tools to monitor the services

You won't need these right now, as you saw the services running, but leaving this here as it is helpful tools in the future:

```
# Watch service status (Ctrl + C to exit)
watch sudo docker service ls

# View service logs
sudo docker service logs myappname_web
sudo docker service logs myappname_viz
```

"Watch" is live monitoring, so you will have to Ctrl + C to exit / cancel the operation.

Logs are useful tools, as a developer these are your best friends. Logging separate logfiles for separate services / functions is something you will want to do and learn.

2.5.4 Web-Summary

We visit the public IPs, check that the service is running and try the Visualizer.

2.6 Tools for cleaning up (optional)

At this point you might want to clean up a stack (if you are running one), or leave / reset the Swarm.

2.6.1 Remove stack

On the manager node: `sudo docker stack rm myappname`

rm means "remove", it is a good command to know if you are working with files and directories too.

2.6.2 Leave Swarm

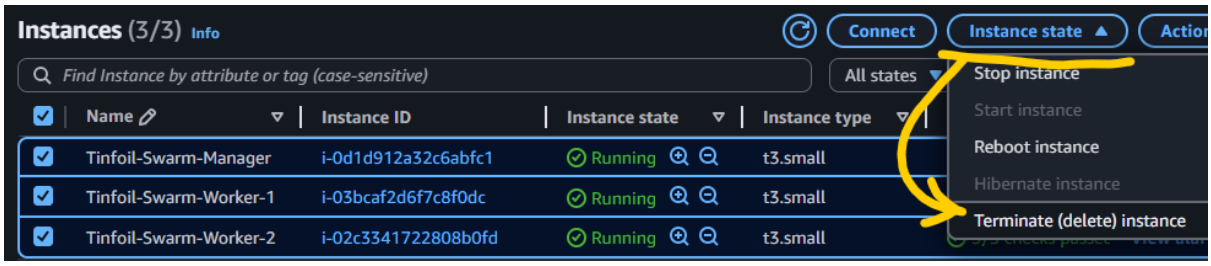
You might want to reset the swarm, alter instances or something else:

```
# On workers
sudo docker swarm leave
```

```
# On manager (force)
sudo docker swarm leave --force
```

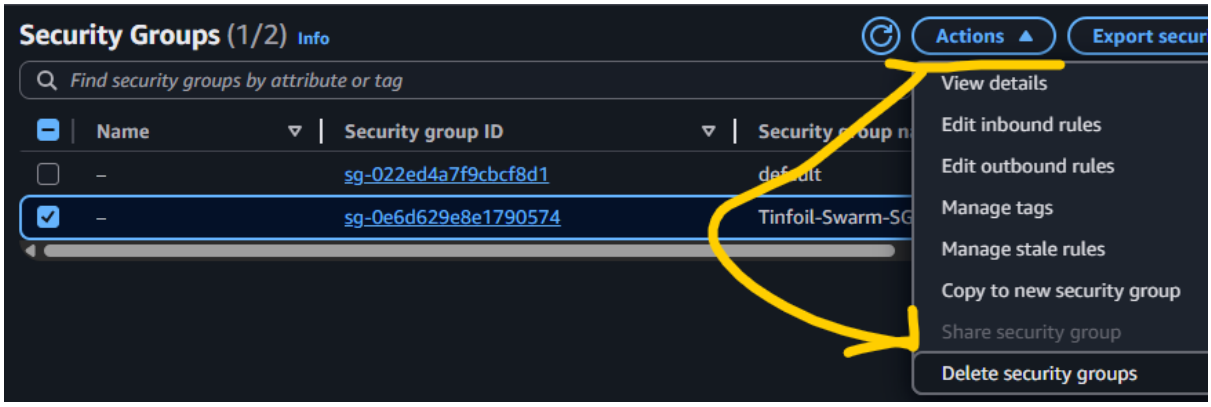
2.6.3 Terminate Instances

Visit EC2 > Instances. Select all three instances. Under “Instance state” chose Terminate.



2.6.4 Delete Security Group

Visit EC2 > Security Groups. Under Actions, Delete the SG.



2.7 Summary of above sections

We have built a SG, launched three instances and deployed a Docker Swarm on them, making sure it is healthy, and all the roles are functional.

In the class when we were at this point, we were supposed to do a .NET application and use DynamoDB for handling files that we send (and possibly retrieve, but that was a bonus) from our .NET application. As I have taken down and up these swarms now multiple times, I feel it is appropriate to create section 3 here – create the above system but in CloudFormation (or Terraform, but let us begin with CF).

3 CloudFormation me up, Scotty

I will build this using the Nested Stacks system as I really, strongly dislike messy long files. It has other benefits too:

- Separations of Concern – Each section is independent from other sections. This is good both from a modularity concept but it also easier when trying to solve problems that arise
- Deploy – Deployment benefits greatly from being separated. You can work on one module and just deploy that one, especially if we are working idempotent
- Adaptability – We might want to expand, or reduce, the program. Adapt to various needs. Then it might be beneficial to just replace or remove or add sections that we need.

That said, let us get going!

3.1 S3 Bucket roles first

Before CloudFormation can deploy nested stacks, we need a storage location for the templates and a user that has permissions to interact with that location. This is done in two parts.

3.1.1 Create the bucket first

1. Go to AWS Console > S3 > Create bucket
2. Enter a unique bucket name (ex., cf-artifacts-<account-id>-eu-west-1). Bucket names must be globally unique
3. Select the region where you plan to deploy your stacks (ex.., eu-west-1)
4. Leave other settings at defaults for now
5. Click Create bucket

You now have a dedicated bucket for storing CloudFormation templates and artifacts.

3.1.2 Then attach the policy to the user so you can use the bucket

1. Go to AWS Console > IAM > Users > select your user
2. Click Add Permissions > Create inline policy
3. Switch to the JSON tab and paste in your policy (replace <your-bucket-name> with the one you just created in 3.1.1.

```
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Effect": "Allow",
      "Action": [
        "s3:ListBucket",
        "s3:GetBucketLocation"
      ],
      "Resource": "arn:aws:s3:::<your-bucket-name>"
    },
    {
      "Effect": "Allow",
      "Action": [
        "s3:GetObject",
        "s3:PutObject",
        "s3:DeleteObject",
        "s3:DeleteObjectVersion"
      ],
      "Resource": "arn:aws:s3:::<your-bucket-name>/*"
```

```
    }
  ]
}
```

Name the policy something descriptive (ex. S3_bucket_policy or cf-s3access-policy) and save it. Your IAM user now has the necessary permissions to interact with the S3 bucket – congrats!

3.2 Quick conversion – CloudFormation style

Before we write anything further: this document has the potential to become rather long from here on. So I will handle it this way – the skeleton of a nested stacks CloudFormation for the above project (section 1 + 2) will be posted here in plain text. But those documents will be growing, and we will add features.

That will too long to handle in text, so I recommend visiting the repo for the latest code:

https://github.com/mymh13/swarm-dotnet-test

3.2.1 Nested stacks structure

Child templates live in S3 and a tiny **root** template orchestrates them in order.

```
/infra
  /templates
    root.yaml                  # orchestrates children
    00-sg-swarm.yaml           # Security Group for swarm
    10-ec2-swarm.yaml          # EC2 Manager + 2 Workers (Docker installed)
  /parameters
    dev.json                   # example parameter set for root.yaml
```

3.2.2 Upload child templates to S3

We upload the SG and EC2-template stacks to our S3-bucket. root has defined URLs to the template stacks. When we run the aws update-stack-command on root it will update with the latest files that we have in the bucket. This way we can do manal work on separate stacks, but the root will orchestrate. This also means we can deploy and take down the entire stack with the root.

3.2.3 root.yaml (Master stack)

This will be a long “skeleton” code, and this all assumes you have basic understanding of AWS and Cloudformation. The SG/EC2-setup we run should be self-explanatory more or less though. Template:

```
AWSTemplateFormatVersion: '2010-09-09'
Description: Root stack for Docker Swarm (manager + 2 workers) using nested
stacks

Parameters:
  StackNamePrefix:
    Type: String
    Default: swarm-cf
    Description: Prefix used for naming
  TemplateBucket:
```

```
    Type: String
    Description: S3 bucket that stores child templates (e.g. cf-artifacts-
<acct>-<region>)
  TemplatePrefix:
    Type: String
    Default: swarm-iac/templates/
    Description: Key prefix inside the S3 bucket
  VpcId:
    Type: AWS::EC2::VPC::Id
    Description: VPC to deploy into (choose your default VPC)
  PublicSubnetId:
    Type: AWS::EC2::Subnet::Id
    Description: Public subnet for all three instances (default subnet is
fine)
  AllowedSshCidr:
    Type: String
    Description: Your IP in CIDR for SSH (e.g. 1.2.3.4/32)
  KeyPairName:
    Type: AWS::EC2::KeyPair::KeyName
    Description: Existing EC2 key pair name
  InstanceType:
    Type: String
    Default: t3.small
  AmiId:
    Type: AWS::EC2::Image::Id
    Description: Amazon Linux 2023 AMI in this region

Resources:
  SwarmSecurityGroupStack:
    Type: AWS::CloudFormation::Stack
    Properties:
      TemplateURL: !Sub
'https://s3.${AWS::Region}.amazonaws.com/${TemplateBucket}/${TemplatePrefix}00
-sg-swarm.yaml'
      Parameters:
        StackNamePrefix: !Ref StackNamePrefix
        VpcId: !Ref VpcId
        AllowedSshCidr: !Ref AllowedSshCidr

  SwarmEc2Stack:
    Type: AWS::CloudFormation::Stack
    DependsOn: SwarmSecurityGroupStack
    Properties:
      TemplateURL: !Sub
'https://s3.${AWS::Region}.amazonaws.com/${TemplateBucket}/${TemplatePrefix}10
-ec2-swarm.yaml'
      Parameters:
        StackNamePrefix: !Ref StackNamePrefix
```

```
        PublicSubnetId: !Ref PublicSubnetId
        SecurityGroupId: !GetAtt
SwarmSecurityGroupStack.Outputs.SwarmSecurityGroupId
        KeyPairName: !Ref KeyPairName
        InstanceType: !Ref InstanceType
        AmiId: !Ref AmiId

Outputs:
  ManagerPublicIp:
    Value: !GetAtt SwarmEc2Stack.Outputs.ManagerPublicIp
  Worker1PublicIp:
    Value: !GetAtt SwarmEc2Stack.Outputs.Worker1PublicIp
  Worker2PublicIp:
    Value: !GetAtt SwarmEc2Stack.Outputs.Worker2PublicIp
  SecurityGroupId:
    Value: !GetAtt SwarmSecurityGroupStack.Outputs.SwarmSecurityGroupId
```

3.2.4 00-sg-swarm.yaml (Swarm Security Group)

This stack controls the Swarms SG.

```
AWSTemplateFormatVersion: '2010-09-09'
Description: Security Group for Swarm (SSH, HTTP, Viz, Swarm ports)

Parameters:
  StackNamePrefix:
    Type: String
  VpcId:
    Type: AWS::EC2::VPC::Id
  AllowedSshCidr:
    Type: String

Resources:
  SwarmSG:
    Type: AWS::EC2::SecurityGroup
    Properties:
      GroupDescription: !Sub '${StackNamePrefix}-swarm-sg'
      VpcId: !Ref VpcId
      SecurityGroupIngress:
        # SSH (locked to your IP)
        - IpProtocol: tcp
          FromPort: 22
          ToPort: 22
          CidrIp: !Ref AllowedSshCidr
        # HTTP 80 (world)
        - IpProtocol: tcp
          FromPort: 80
          ToPort: 80
```

```
          CidrIp: 0.0.0.0/0
        # Visualizer 8080 (world)
        - IpProtocol: tcp
          FromPort: 8080
          ToPort: 8080
          CidrIp: 0.0.0.0/0
        # Swarm mgmt 2377 (self-reference)
        - IpProtocol: tcp
          FromPort: 2377
          ToPort: 2377
          SourceSecurityGroupId: !Ref SwarmSG
        # Gossip TCP 7946 (self-reference)
        - IpProtocol: tcp
          FromPort: 7946
          ToPort: 7946
          SourceSecurityGroupId: !Ref SwarmSG
        # Gossip UDP 7946 (self-reference)
        - IpProtocol: udp
          FromPort: 7946
          ToPort: 7946
          SourceSecurityGroupId: !Ref SwarmSG
        # Overlay UDP 4789 (self-reference)
        - IpProtocol: udp
          FromPort: 4789
          ToPort: 4789
          SourceSecurityGroupId: !Ref SwarmSG
      Tags:
        - Key: Name
          Value: !Sub '${StackNamePrefix}-swarm-sg'

Outputs:
  SwarmSecurityGroupId:
    Value: !Ref SwarmSG
```

3.2.5 10-ec2-swarm.yaml (EC2 Manager + 2 Workers, Docker installed)

Here we create the EC2-Swarm, and set the User Data to install docker, like we did here.

```
AWSTemplateFormatVersion: '2010-09-09'
Description: EC2 instances for Swarm (1 manager, 2 workers) with Docker
installed

Parameters:
  StackNamePrefix:
    Type: String
  PublicSubnetId:
    Type: AWS::EC2::Subnet::Id
  SecurityGroupId:
```

```
    Type: String
  KeyPairName:
    Type: AWS::EC2::KeyPair::KeyName
  InstanceType:
    Type: String
  AmiId:
    Type: AWS::EC2::Image::Id

Mappings: {}

Resources:
  Manager:
    Type: AWS::EC2::Instance
    Properties:
      ImageId: !Ref AmiId
      InstanceType: !Ref InstanceType
      KeyName: !Ref KeyPairName
      SubnetId: !Ref PublicSubnetId
      SecurityGroupIds: [ !Ref SecurityGroupId ]
      Tags:
        - Key: Name
          Value: !Sub '${StackNamePrefix}-manager'
      UserData:
        Fn::Base64: !Sub |
          #cloud-config
          runcmd:
            - dnf update -y
            - dnf install -y docker
            - systemctl enable --now docker
            - usermod -aG docker ec2-user

  Worker1:
    Type: AWS::EC2::Instance
    Properties:
      ImageId: !Ref AmiId
      InstanceType: !Ref InstanceType
      KeyName: !Ref KeyPairName
      SubnetId: !Ref PublicSubnetId
      SecurityGroupIds: [ !Ref SecurityGroupId ]
      Tags:
        - Key: Name
          Value: !Sub '${StackNamePrefix}-worker-1'
      UserData:
        Fn::Base64: !Sub |
          #cloud-config
          runcmd:
            - dnf update -y
            - dnf install -y docker
```

```
            - systemctl enable --now docker
            - usermod -aG docker ec2-user

  Worker2:
    Type: AWS::EC2::Instance
    Properties:
      ImageId: !Ref AmiId
      InstanceType: !Ref InstanceType
      KeyName: !Ref KeyPairName
      SubnetId: !Ref PublicSubnetId
      SecurityGroupIds: [ !Ref SecurityGroupId ]
      Tags:
        - Key: Name
          Value: !Sub '${StackNamePrefix}-worker-2'
      UserData:
        Fn::Base64: !Sub |
          #cloud-config
          runcmd:
            - dnf update -y
            - dnf install -y docker
            - systemctl enable --now docker
            - usermod -aG docker ec2-user

Outputs:
  ManagerPublicIp:
    Value: !GetAtt Manager.PublicIp
  Worker1PublicIp:
    Value: !GetAtt Worker1.PublicIp
  Worker2PublicIp:
    Value: !GetAtt Worker2.PublicIp
```

Note:
This intentionally stops at Docker installed. After stack completes, follow the tutorial's SSH steps:

- SSH to manager, run docker swarm init --advertise-addr <manager-private-ip>.
- SSH to each worker, run the docker swarm join ... command from the manager output.
- Deploy your docker-stack.yml exactly as before.

3.2.6 /parameters/dev.json (example)

This is just a brief section that introduces the dev.json parameters.

X. References

X.1 educ8.se

The basics of this guide is based on study material and tutorials handed out during the Cloud Developer YH-program (school of applied knowledge) at Campus Mölndal in 2025. Copyright to that material belongs to our teacher Lars Appel, this guide has been modified (and I have taken personal design choices) to reflect this. Lars' guides are more handing out the basics, this one will try to explain why we chose what. See more at: https://educ8me.se

X.2