AWS Notes

Table of Contents

AWS -- Amazon Web Services

A large collection of cloud based services from Amazon.

A single account enables access to all AWS services, you pay for resources you use.

- https://aws.amazon.com -- Sign up here.
- https://console.aws.amazon.com/console/ -- The Web Console, for all AWS Services

AWS CLI

The AWS Command Line Interface (CLI) is a unified tool to manage your AWS services. With just one tool to download and configure, you can control multiple AWS services from the command line and automate them through scripts.

- The AWS Web Console rapidly evolves (changes all the time), but the AWS CLI is more stable, use it when you can!
- Install and use aws-shell! Super useful with the CLI.
- The AWS CLI signs requests on your behalf, and includes a date in the signature. Ensure that your computer's date and time are set correctly;

We will be using CLI commands for the rest of this document.

IAM -- Identity and Access Management

https://console.aws.amazon.com/iam/

Creating Users, Groups

Initially, right after signing up on AWS as the "root" user, you should create one or more non-root users to be used for day-to-day AWS access.

Example commands:

- `aws iam create-group --group-name Admins` -- Create the group `Admins`

- `aws iam attach-group-policy --group-name Admins --policy-arn arn:aws:iam::aws:policy/AdministratorAccess` -- Attach the `AdministratorAccess` policy (predetermined by AWS) to the group.

- `aws iam create-user --user-name admin` -- Create the user `admin`

- `aws iam create-login-profile --user-name admin --password MySecurePasswordWithSpecialCH#R$` -- Add a password to the user that enables him to log into the web console. Use the `update-login-profile` command to update the password for an a user.

- `aws iam add-user-to-group --group-name Admins --user-name admin` -- Add the `admin` user to the `Admins` group.

- `aws> iam create-access-key --user-name admin` -- Creates user credentials. Warning: The secret access key is accessible only during key and user creation. You must save the key (for example, in a text file) if you want to be able to access it again. If a secret key is lost, you can delete the access keys for the associated user and then create new keys. **SAVE the `AccessKeyId` and `SecretAccessKey` values!**

- `aws configure` -- Configures your workstation with your credentials (you enter them when prompted).

```
$ aws configure
AWS Access Key ID [None]: AKIAIOSFODNN7EXAMPLE
AWS Secret Access Key [None]: wJalrXUtnFEMI/K7MDENG/bPxRfiCYEXAMPLEKEY
Default region name [None]: us-west-2
Default output format [None]: json
```

- This point forward, the AWS CLI will use these credentials to interact with the service. These credentials DO NOT EXPIRE. So, **how to "log out" of AWS on your workstation?**
    - Either overwrite the "configuration" with running the `aws configure` command again and enter null/empty values and save this configuration.
    - Or delete your `~/.aws/credentials` file on your workstation.

Security Group

https://docs.aws.amazon.com/AWSEC2/latest/UserGuide/using-network-security.html?icmpid=docs_ec2_console

A security group acts as a virtual firewall that controls the traffic for one or more EC2 instances. When you launch an instance, you associate one or more security groups with the instance. You add rules to each security group that allow traffic to or from its associated instances.

You can modify the rules for a security group at any time; the new rules are automatically applied to all instances that are associated with the security group. When we decide whether to allow traffic to reach an instance, we evaluate all the rules from all the security groups that are associated with the instance.

Example commands:

- `aws ec2 create-security-group --group-name aws-sec-grp-us-east1 --description "Security Group for admin on US. East1"`
- `aws ec2 describe-security-groups --group-id sg-23b7064b` -- Verify, from the output of the previous command.
- `aws ec2 authorize-security-group-ingress --group-id sg-23b7064b --protocol tcp --port 22 --cidr 0.0.0.0` -- Allow inbound TCP traffic on port 22
- `aws ec2 authorize-security-group-ingress --group-id sg-23b7064b --protocol tcp --port 80 --cidr 0.0.0.0` -- Allow inbound TCP traffic on port 80
- `aws ec2 authorize-security-group-ingress --group-id sg-23b7064b --protocol tcp --port 5432 --cidr 0.0.0.0 --source-group sg-23b7064b` -- Allow inbound TCP traffic on port 5432 (Postgres), but ONLY originated from instances inside the same security group.
- `aws ec2 authorize-security-group-ingress --group-id sg-23b7064b --protocol tcp --port 6379 --cidr 0.0.0.0 --source-group sg-23b7064b` -- Allow inbound TCP traffic on port 6739 (Redis), within the same security group only.
- `aws ec2 describe-security-groups --group-id sg-23b7064b` -- Lists permissions for the security group.

Roles

IAM roles are a secure way to grant permissions to entities that you trust on AWS resources. Examples of entities include the following:

- Application code running on an EC2 instance that needs to perform actions on AWS resources
- IAM user in another account

- An AWS service that needs to act on resources in your account to provide its features
- Users from a corporate directory who use identity federation with SAML

IAM roles issue keys that are valid for short duration, making them a more secure way to grant access.

It is too difficult to create roles with the CLI, it would require passing in JSON files as "trust policies" that I could not find anywhere, so here are the instructions using the Web Console.

1. Go to the IAM Console
2. Create New Role (for EC2 instances to join ECS Cluster, and access S3 file storage):

- Type AWS Service,
- Service: EC2 Container Service,
- Use case: EC2 Role for EC2 Container Service,
- Name: ecsInstanceRole

3. Attach one more policy: AmazonS3readOnlyAccess, to the ecsInstanceRole Role.
4. Create Another Role (to handle cases when another AWS service will need to make API calls on our behalf. In this case the ELB (load balancers) will use it to interact with ECS Service when deciding about nodes being up/down):

- Type AWS Service,
- Service: EC2 Container Service,
- Use case: EC2 Container Service,
- Name: ecsServiceRole

These roles will be used when creating EC2 instances for our ECS Services.

EC2 -- Elastic Compute Cloud

Virtual Machines for rent. ECS uses EC2 instances to run Container instances.

Example commands to create SSH key pairs for accessing EC2 Instances:

- ec2 create-key-pair --key-name aws-admin --query 'KeyMaterial' --output text > ~/.ssh/aws-admin.pem -- On Linux
- chmod 400 ~/.ssh/aws-admin.pem -- On Linux
- ec2 create-key-pair --key-name aws-admin --query 'KeyMaterial' --output text | out-file -encoding ascii -filepath ./.ssh/aws-admin.pem -- On Windows. Might need to move the file into ~/.ssh folder
- aws ec2 describe-key-pairs --key-name aws-admin - Verify that the key pair has been created.
- This is the kay pair we are going to be using when creating EC2 Instances, so we can SSH into them.
- On Windows, if using PUTTY to SSH into the VM, this is the PEM file that you want to specify on Connection/SSH/Auth/PrivateKey.

ELB -- Elastic Load Balancing

- https://aws.amazon.com/documentation/elastic-load-balancing/
- https://docs.aws.amazon.com/elasticloadbalancing/latest/userguide/what-is-load-balancing.html?icmpid=docs_elbv2_console

Elastic Load Balancing distributes incoming application traffic across multiple EC2 instances, in multiple Availability Zones. This increases the fault tolerance of your applications.

The load balancer serves as a single point of contact for clients, which increases the availability of your application. You can add and remove instances from your load balancer as your needs change, without disrupting the overall flow of requests to your application. Elastic Load Balancing scales your load balancer as traffic to your application changes over time, and can scale to the vast majority of workloads automatically.

You can configure health checks, which are used to monitor the health of the registered instances so that the load balancer can send requests only to the healthy instances. You can also offload the work of encryption and decryption to your load balancer so that your instances can focus on their main work.

ECR -- EC2 Container Registry

ECR is managed Docker Registry.

It's part of ECS, but in the CLI it is a separate "service".

Example commands:

- `aws ecr get-login` -- Generates a `docker login` command that logs you into the AWS Container Registry for 12 hours.
- `aws ecr create-repository --repository-name deepdive/nginx` -- Create a new repository
- `aws ecr describe-repositories` -- Describe all repositories
- `aws ecr list-images --repository-name deepdive/nginx` -- List all images in the deepdive/nginx repository
- `docker pull nginx:1.9` -- Pull down the nginx image from DockerHub, so we can push it later
- `docker tag nginx:1.9 xxx.dkr.ecr.us-east-1.amazonaws.com/deepdive/nginx:1.9` -- Tag the nginx image locally
- `docker push xxx.dkr.ecr.us-east-1.amazonaws.com/deepdive/nginx` -- Push the nginx image to the ECR repository

ECS -- EC2 Container Service

ECS CLI

- Amazon ECS CLI on GitHub
- Supports working with `docker-compose` files. Not available on Windows.
- Amazon's tutorial on the ECS CLI

Clusters

A *Cluster* is a group of container instances that act as a single computing resource.

There is the default cluster, if you don't specify a cluster name.

Example commands:

- aws ecs create-cluster --cluster-name deepdive -- Create the 'deepdive' cluster
- aws ecs list-clusters
- aws ecs describe-clusters --clusters deepdive
- aws ecs delete-cluster --cluster deepdive

Prepare the S3 bucket for ECS Container Agent configuration

Example commands:

- aws s3api create-bucket --bucket tkarakai-gto-deepdive -- Create the S3 bucket for the deepdive cluster
- aws s3 cp ecs.config s3://tkarakai-gto-deepdive/ecs.config -- Copy the ECS config file to the S3 bucket:

```
ECS_CLUSTER=deepdive
```

- aws s3 ls s3://tkarakai-gto-deepdive -- Verify the ECS config is in the S3 bucket

ECS Container Agent

The *Container Agent* is a tool installed on the EC2 Instances that allows the instance to join the ECS Cluster.

- Written in GoLang
- Open Source, available on GitHub
- There is a Docker Image for it
- There are ECS Optimized AMIs per region that we are free to use. It is Amazon tested, contains a not too old version od Docker installed and the ECS Container Agent (as a Docker Container). Take their AMI ID to be used in the aws ec2 run-instances --image-id [...] ... command.
- Lots of configurations, most of which are via environment variables.
- If your container instance was launched **with the Amazon ECS-optimized AMI**, you can set these environment variables in the /etc/ecs/ecs.config file and then restart the agent.
- **If you are manually starting** the Amazon ECS container agent (for non-Amazon ECS-optimized AMIs), you can use these environment variables in the docker run command that you use to start the agent with the syntax --env=VARIABLE_NAME=VARIABLE_VALUE.

Container Instances

Container Instance is an EC2 Instance which is part of a Cluster, not to be confused with a *Container* that is a Docker Container started via a *Task*.

Example commands:

- aws ec2 run-instances --image-id ami-1c002379 --count 3 --instance-type t2.micro --iam-instance-profile Name=ecsInstanceRole --key-name aws-admin --security-group-ids sg-23b7064b --user-data file://copy-ecs-config-to-s3
    - Starts 3 new EC2 instances from the default us-east-2 ESC container AMI (image),
    - with "InstanceRole",
    - having our SSH key installed,
    - controlled by the security group (with specific ports open),
    - with a user script to be executed at instance startup containing the cluster agent configuration.
- aws ec2 describe-instance-status --instance-id i-0c74a0c20ed35fb51 -- Don't forget the --instance-id (from the output of the previous command), otherwise it will return an empty result.
- aws ecs list-container-instances --cluster deepdive --container-instances [Arm from prev cmd output] -- Info on container instances, including things like Docker version running, status (ACTIVE/INACTIVE), running tasks, etc.
- aws ec2 terminate-instances --instance-ids ...

Task Definitions

A *task definition* is a JSON document describing how your applications Docker Images should be ran. It's kind of like a docker-compose file, a la AWS. The application can include one or more Docker container instances. It might associate volumes to the containers. It defines the Docker images, along with settings such as the memory and CPU resources assigned to a container, or the ports mapped to a container.

Example:

File web-task-definition.json:

```
{
  "containerDefinitions": [
    {
      "name": "nginx",
      "image": "514778609496.dkr.ecr.us-east-2.amazonaws.com/dockerzon/nginx",
      "portMappings": [
        {
          "containerPort": 80,
          "hostPort": 80
        }
      ],
      "links": ["dockerzon:dockerzon"],
      "volumesFrom": [
        {
          "sourceContainer": "dockerzon"
        }
      ],
      "cpu": 256,
      "memory": 100
    },
    {
      "name": "dockerzon",
      "image": "514778609496.dkr.ecr.us-east-
2.amazonaws.com/dockerzon/dockerzon",
      "portMappings": [
        {
          "containerPort": 8000,
          "hostPort": 8000
        }
      ],
      "cpu": 768,
      "memory": 300,
      "environment": [
        {
          "name": "RAILS_ENV",
          "value": "production"
        },
        {
          "name": "DATABASE_URL",
          "value":
"postgresql://dockerzon:intergalacticzebramoldfactory@dockerzon-
production.csyqs7vrgsil.us-east-2.rds.amazonaws.com:5432/dockerzon?
encoding=utf8&pool=40&timeout=5000"
        },
        {
          "name": "CACHE_URL",
          "value": "redis://dockerzon-
production.scz2ms.0001.use2.cache.amazonaws.com:6379"
        },
        {
          "name": "JOB_WORKER_URL",
          "value": "redis://dockerzon-
production.scz2ms.0001.use2.cache.amazonaws.com:6379"
        },
        {
          "name": "SECRET_TOKEN",
          "value":
"43d2aca242bf88ddb1ea4ef8be8be38b754d6cb6314931cb9b648723e82e9a203232f377ae43538a
55059b09c419108f0312e1b0704a8cf5eb1ef4018a6ab924"
        }
      ]
    }
  ],
  "family": "web"
  }
```

Example commands:

- aws ecs register-task-definition --cli-input-json file://web-task-definition.json
- aws ecs list-task-definition-families
- aws ecs describe-container-instances --cluster deepdive --container-instances arn:aws:ecs:us-east-2:514778609496:container-instance/aecd6737-cc3f-467c-8115-6a27474262c1

Running Tasks

A *Task* the end result of running a task definition. Running a task is comparable to a *service*, but with the task, the containers do not need to be automatically restarted when stopped (a service will try to keep tasks running). It is useful to execute one-off task, like migrations or other batch jobs.

The run-task command randomly distributes the tasks among the container instances in the cluster, trying to minimize resource overload.

The aws start-task command runs a task on a particular instance or instances (useful when certain instances are better suited for the task, e.g. have more memory or CPUs).

Service and task life cycle (as tracked and reported by the *Container Agent*):

- PENDING
- RUNNING
- STOPPED

Example commands:

- aws ecs run-task --cluster deepdive --task-definition web --count 1
- aws ecs list-tasks --cluster deepdive
- aws ecs stop-task --cluster development --task [full ARN of the task from prev cmd output]
- aws ecs list-container-instances --cluster deepdive
- aws ecs start-task --cluster deepdive --task-definition web --container-instances [ARN of instance from prev cmd output]
- aws ecs stop-task --cluster deepdive --task [task ARN]

Scheduling Services

A *service* consists of a certain number of a long running task. This enables both availability as well as scalability to your application. As an example, imagine a web application, which consists of different instances of the web server running on different container instances.

A service also helps with the monitoring of existing tasks. Should one task fail or stop running due to whatever reason, a new task is automatically started up again.

The *scheduler* determines where a service or a task will run on a cluster by figuring out the most optimal instance to run it on.

Decisions the *scheduler* will make to place a task into a cluster:

1. Collect the service requirements (CPU, Memory, ports) and collect stats about currently running service tasks in all Container Instances in all availability zones.
2. Selects the best Availability Zone (with the most remaining resources)
3. Selects the best Container Instance within the AZ (with the most remaining resources)

Example commands:

- `aws ecs create-service --cluster deepdive --service-name web --task-definition web --desired-count 1` -- Create service without a Service Definition file, no load balancers
- `aws ecs list-services --cluster deepdive`
- `aws ecs describe-services --cluster deepdive --services web` -- Look for the `events` section when troubleshooting!
- `aws ec2 describe-instances` - to get to know the public DNS name of the service
- `aws ecs update-service --cluster deepdive --service web --task-definition web --desired-count 2` -- More instances (will fail if there are not enough instances, and pointless without load balancing!)
- `aws ecs update-service --cluster deepdive --service web --task-definition web --desired-count 0` -- No instances!
- `aws ecs delete-service --cluster deepdive --service web`
- `aws ecs list-services --cluster deepdive`
- `aws ecs create-service --generate-cli-skeleton` -- Generates a JSON template with all possible parameters to set for a Service Definition:

```
{
    "cluster": "",
    "serviceName": "",
    "taskDefinition": "",
    "loadBalancers": [
        {
            "targetGroupArn": "",
            "loadBalancerName": "",
            "containerName": "",
            "containerPort": 0
        }
    ],
    "desiredCount": 0,
    "clientToken": "",
    "role": "",
    "deploymentConfiguration": {
        "maximumPercent": 0,
        "minimumHealthyPercent": 0
    },
    "placementConstraints": [
        {
            "type": "memberOf",
            "expression": ""
        }
    ],
    "placementStrategy": [
        {
            "type": "spread",
            "field": ""
        }
    ]
}
```

The Service Definition defines which task definition to use with your service, how many instantiations of that task to run, and which load balancers (if any) to associate with your tasks.

A service can be "hooked up" to an *ELB* (Elastic Load Balancer) where ELB handles adding and removing instances automatically when the service scales up/down or when an instance becomes unhealthy.

Example:

File web-service.json:

```
{
    "cluster": "production",
    "serviceName": "web",
    "taskDefinition": "web",
    "loadBalancers": [
        {
            "loadBalancerName": "dockerzon-web",
            "containerName": "nginx",
            "containerPort": 80
        }
    ],
    "role": "escServiceRole",
    "desiredCount": 2,
    "deploymentConfiguration": {
        "maximumPercent": 100,
        "minimumHealthyPercent": 50
    }
}
```

Tearing Down a Cluster

- `aws ec2 terminate-instances --instance-ids i-0c74a0c20ed35fb51`
- `aws s3 rm s3://tkarakai-gto-test --recursive`
- `aws s3api delete-bucket --bucket tkarakai-gto-test`

Ruby on Rails App example

- `docker run --rm --user "$(id -u):$(id -g)" -v "$PWD":/usr/src/app -w /usr/src/app rails:4 rails new --skip-bundle dockerzon`

AWS Terms (Glossary)

- S3 -- Simple Storage Service
- Route 53 -- DNS Service
- RDS -- Relational Database Service
  - Engines supported:
    - MySQL
    - MariaDB
    - Postgres
    - MsSQL
    - Oracle
    - Amazon Aurora (not eligable on free tier)
- ElastiCache -- Managed In-Memory data store and cache
- Engines supported:
  - Redis
  - Memcached
- AMI -- Amazon Machine Image
- ARN -- Amazon Resource Name

F.A.Q., Troubleshooting

PROBLEM: When you issue `aws s3api create-bucket --bucket tkarakai-gto-dockerzon` and it results in the following error: `An error occurred (IllegalLocationConstraintException) when calling the CreateBucket operation: The unspecified location constraint is incompatible for the region specific endpoint this request was sent to.`,

SOLUTION: ...just add the location constraint to the command like this: `aws s3api create-bucket --bucket tkarakai-gto-dockerzon --region us-east-2 --create-bucket-configuration LocationConstraint=us-east-2`

PROBLEM: How do I add instances to a load balancer?

SOLUTION:

PROBLEM: How do I start tasks on a cluster?

SOLUTION:

PROBLEM: How does the aws ec2 run-instances --image-id ami-... command knows to join an ECS cluster?

SOLUTION: Based on the ecs.config file, in there the ECS_CLUSTER=production configuration (in this case specifying the cluster called production). This file is made available by copying it in to the instance from an S3 bucket. The script that copies it in is specified in the --user-data file://copy-ecs-config-to-s3 which gets executed on the instance at startup time.

File: copy-ecs-config-to-s3

```
#!/bin/bash

yum install -y aws-cli
aws s3 cp s3://tkarakai-gto-dockerzon/ecs.config /etc/ecs/ecs.config
```

File: ecs.config

```
ECS_CLUSTER=production
```
