**AWS KT Document**

**EC2 Instances in AWS**

Open the Amazon EC2 console at https://console.aws.amazon.com/ec2/.

Choose Launch Instance.

In Step 1: Choose an Amazon Machine Image (AMI), find an Amazon Linux 2 AMI at the top of the list and choose Select.

In Step 2: Choose an Instance Type, choose Next: Configure Instance Details.

In Step 3: Configure Instance Details, provide the following information:

Leave Number of instances at one.

Leave Purchasing option at the default setting.

For Network, choose the entry for the same VPC that you noted when you created your EFS file system in Step 1: Create your Amazon EFS file system.

For Subnet, choose a default subnet in any Availability Zone.

For File systems, make sure that the EFS file system that you created in Step 1: Create your Amazon EFS file system is selected. The path shown next to the file system ID is the mount point that the EC2 instance will use, which you can change.

The User data automatically includes the commands for mounting your Amazon EFS file system.

Choose Next: Add Storage.

Choose Next: Add Tags.

Name your instance and choose Next: Configure Security Group.

In Step 6: Configure Security Group, set Assign a security group to Select an existing security group. Choose the default security group to make sure that it can access your EFS file system.

Choose Review and Launch.

Choose Launch.

Select the check box for the key pair that you created, and then choose Launch Instances.

Once the EC2 instance is created and becomes available, it will be mounted to your EFS file system. At this point, you will be able to transfer files to your EFS file system.

**Creating, configuring, and working with Amazon S3 buckets**

To store your data in Amazon S3, you work with resources known as buckets and objects. A bucket is a container for objects. An object is a file and any metadata that describes that file.

To store an object in Amazon S3, you create a bucket and then upload the object to a bucket. When the object is in the bucket, you can open it, download it, and move it. When you no longer need an object or a bucket, you can clean up your resources.

To upload your data (photos, videos, documents, etc.) to Amazon S3, you must first create an S3 bucket in one of the AWS Regions.

A bucket is a container for objects stored in Amazon S3. You can store any number of objects in a bucket and can have up to 100 buckets in your account. To request an increase, visit the Service Quotas Console.

Every object is contained in a bucket. For example, if the object named photos/puppy.jpg is stored in the DOC-EXAMPLE-BUCKET bucket in the US West (Oregon) Region, then it is addressable using the URL https://DOC-EXAMPLE-BUCKET.s3.us-west-2.amazonaws.com/photos/puppy.jpg. For more information, see Accessing a Bucket.

In terms of implementation, buckets and objects are AWS resources, and Amazon S3 provides APIs for you to manage them. For example, you can create a bucket and upload objects using the Amazon S3 API. You can also use the Amazon S3 console to perform these operations. The console uses the Amazon S3 APIs to send requests to Amazon S3.

This section describes how to work with buckets. For information about working with objects, see Amazon S3 objects overview.

Amazon S3 supports global buckets, which means that each bucket name must be unique across all AWS accounts in all the AWS Regions within a partition. A partition is a grouping of Regions. AWS currently has three partitions: aws (Standard Regions), aws-cn (China Regions), and aws-us-gov (AWS GovCloud (US)).

After a bucket is created, the name of that bucket cannot be used by another AWS account in the same partition until the bucket is deleted. You should not depend on specific bucket naming conventions for availability or security verification purposes. For bucket naming guidelines, see Bucket naming rules.

Amazon S3 creates buckets in a Region that you specify. To optimize latency, minimize costs, or address regulatory requirements, choose any AWS Region that is geographically close to you. For example, if you reside in Europe, you might find it advantageous to create buckets in the Europe (Ireland) or Europe (Frankfurt) Regions. For a list of Amazon S3 Regions, see Regions and Endpoints in the AWS General Reference.

**AWS Identity and Access Management (IAM)**

AWS Identity and Access Management (IAM) is a web service that helps you securely control access to AWS resources. You use IAM to control who is authenticated (signed in) and authorized (has permissions) to use resources.

When you first create an AWS account, you begin with a single sign-in identity that has complete access to all AWS services and resources in the account. This identity is called the AWS account root user and is accessed by signing in with the email address and password that you used to create the account. We strongly recommend that you do not use the root user for your everyday tasks, even the administrative ones. Instead, adhere to the best practice of using the root user only to create your first IAM user. Then securely lock away the root user credentials and use them to perform only a few account and service management tasks.

IAM features

IAM gives you the following features:

**Shared access to your AWS account**

You can grant other people permission to administer and use resources in your AWS account without having to share your password or access key.

**Granular permissions**

You can grant different permissions to different people for different resources. For example, you might allow some users complete access to Amazon Elastic Compute Cloud (Amazon EC2), Amazon Simple Storage Service (Amazon S3), Amazon DynamoDB, Amazon Redshift, and other AWS services. For other users, you can allow read-only access to just some S3 buckets, or permission to administer just some EC2 instances, or to access your billing information but nothing else.

**Secure access to AWS resources for applications that run on Amazon EC2**

You can use IAM features to securely provide credentials for applications that run on EC2 instances. These credentials provide permissions for your application to access other AWS resources. Examples include S3 buckets and DynamoDB tables.

**Multi-factor authentication (MFA)**

You can add two-factor authentication to your account and to individual users for extra security. With MFA you or your users must provide not only a password or access key to work with your account, but also a code from a specially configured device.

**Identity federation**

You can allow users who already have passwords elsewhere—for example, in your corporate network or with an internet identity provider—to get temporary access to your AWS account.

**Identity information for assurance**

If you use AWS CloudTrail, you receive log records that include information about those who made requests for resources in your account. That information is based on IAM identities.

**PCI DSS Compliance**

IAM supports the processing, storage, and transmission of credit card data by a merchant or service provider, and has been validated as being compliant with Payment Card Industry (PCI) Data Security Standard (DSS). For more information about PCI DSS, including how to request a copy of the AWS PCI Compliance Package, see PCI DSS Level 1.

**Integrated with many AWS services**

For a list of AWS services that work with IAM, see AWS services that work with IAM.

**Eventually Consistent**

IAM, like many other AWS services, is eventually consistent. IAM achieves high availability by replicating data across multiple servers within Amazon's data centers around the world. If a request to change some data is successful, the change is committed and safely stored. However, the change must be replicated across IAM, which can take some time. Such changes include creating or updating users, groups, roles, or policies. We recommend that you do not include such IAM changes in the critical, high-availability code paths of your application. Instead, make IAM changes in a separate initialization or setup routine that you run less frequently. Also, be sure to verify that the changes have been propagated before production workflows depend on them. For more information, see Changes that I make are not always immediately visible.

**Free to use**

AWS Identity and Access Management (IAM) and AWS Security Token Service (AWS STS) are features of your AWS account offered at no additional charge. You are charged only when you access other AWS services using your IAM users or AWS STS temporary security credentials. For information about the pricing of other AWS products, see the Amazon Web Services pricing page.

**Accessing IAM**

You can work with AWS Identity and Access Management in any of the following ways.

**AWS Management Console**

The console is a browser-based interface to manage IAM and AWS resources. For more information about accessing IAM through the console, see Signing in to the AWS Management Console as an IAM user or root user. For a tutorial that guides you through using the console, see Creating your first IAM admin user and user group.

**AWS Command Line Tools**

You can use the AWS command line tools to issue commands at your system's command line to perform IAM and AWS tasks. Using the command line can be faster and more convenient than the console. The command line tools are also useful if you want to build scripts that perform AWS tasks.

AWS provides two sets of command line tools: the AWS Command Line Interface (AWS CLI) and the AWS Tools for Windows PowerShell. For information about installing and using the AWS CLI, see the AWS Command Line Interface User Guide. For information about installing and using the Tools for Windows PowerShell, see the AWS Tools for Windows PowerShell User Guide.

**AWS SDKs**

AWS provides SDKs (software development kits) that consist of libraries and sample code for various programming languages and platforms (Java, Python, Ruby, .NET, iOS, Android, etc.). The SDKs provide a convenient way to create programmatic access to IAM and AWS. For example, the SDKs take care of tasks such as cryptographically signing requests, managing errors, and retrying requests automatically. For information about the AWS SDKs, including how to download and install them, see the Tools for Amazon Web Services page.

**IAM HTTPS API**

You can access IAM and AWS programmatically by using the IAM HTTPS API, which lets you issue HTTPS requests directly to the service. When you use the HTTPS API, you must include code to digitally sign requests using your credentials. For more information, see Calling the IAM API using HTTP query requests and the IAM API Reference.

**Amazon Elastic Container Registry (Amazon ECR)**

Amazon Elastic Container Registry (Amazon ECR) is an AWS managed container image registry service that is secure, scalable, and reliable. Amazon ECR supports private repositories with resource-based permissions using AWS IAM. This is so that specified users or Amazon EC2 instances can access your container repositories and images. You can use your preferred CLI to push, pull, and manage Docker images, Open Container Initiative (OCI) images, and OCI compatible artifacts.

The AWS container services team maintains a public roadmap on GitHub. It contains information about what the teams are working on and allows all AWS customers the ability to give direct feedback. For more information, see AWS Containers Roadmap.

**Components of Amazon ECR**

Amazon ECR contains the following components:

**Registry**

An Amazon ECR private registry is provided to each AWS account; you can create one or more repositories in your registry and store images in them. For more information, see Amazon ECR private registry.

**Authorization token**

Your client must authenticate to Amazon ECR registries as an AWS user before it can push and pull images. For more information, see Private registry authentication.

**Repository**

An Amazon ECR repository contains your Docker images, Open Container Initiative (OCI) images, and OCI compatible artifacts. For more information, see Amazon ECR private repositories.

**Repository policy**

You can control access to your repositories and the images within them with repository policies. For more information, see Private repository policies.

**Image**

You can push and pull container images to your repositories. You can use these images locally on your development system, or you can use them in Amazon ECS task definitions and Amazon EKS pod specifications. For more information, see Using Amazon ECR images with Amazon ECS and Using Amazon ECR Images with Amazon EKS.

**Features of Amazon ECR**

Amazon ECR provides the following features:

Lifecycle policies help with managing the lifecycle of the images in your repositories. You define rules that result in the cleaning up of unused images. You can test rules before applying them to your repository. For more information, see Lifecycle policies.

Image scanning helps in identifying software vulnerabilities in your container images. Each repository can be configured to scan on push. This ensures that each new image pushed to the repository is scanned. You can then retrieve the results of the image scan. For more information, see Image scanning.

Cross-Region and cross-account replication makes it easier for you to have your images where you need them. This is configured as a registry setting and is on a per-Region basis. For more information, see Private registry settings.

Pull through cache rules provide a way to cache repositories in remote public registries in your private Amazon ECR registry. Using a pull through cache rule, Amazon ECR will periodically reach out to the remote registry to ensure the cached image in your Amazon ECR private registry is up to date. For more information, see Using pull through cache rules.

**How to get started with Amazon ECR**

To use Amazon ECR, you must be set up to install the AWS Command Line Interface and Docker. For more information, see Setting up with Amazon ECR and Using Amazon ECR with the AWS CLI.

**Pricing for Amazon ECR**

With Amazon ECR, you only pay for the amount of data you store in your repositories and for the data transfer from your image pushes and pulls.

**Amazon Elastic Container Service (Amazon ECS)**

AWS and Docker have collaborated to make a simplified developer experience that enables you to deploy and manage containers on Amazon ECS directly using Docker tools. You can now build and test your containers locally using Docker Desktop and Docker Compose, and then deploy them to Amazon ECS on Fargate. To get started with the Amazon ECS and Docker integration, download Docker Desktop and optionally sign up for a Docker ID. For more information, see Docker Desktop and Docker ID signup.

Docker provides a walkthrough on deploying containers on Amazon ECS. For more information, see Deploying Docker containers on Amazon ECS.

The documentation in this guide assumes that readers possess a basic understanding of what Docker is and how it works. For more information about Docker, see What is Docker? and the Docker overview.

**Install Docker**

Important

If you already have Docker installed, skip to Create a Docker image.

Docker Desktop is an easy-to-install application for your Mac or Windows environment that enables you to build and share containerized applications and microservices. Docker Desktop includes Docker Engine, the Docker CLI client, Docker Compose, and other tools that are helpful when using Docker with Amazon ECS. For more information about how to install Docker Desktop on your preferred operating system, see Docker Desktop overview.

If you don't need a local development environment and you prefer to use an Amazon EC2 instance to use Docker, we provide the following steps to launch an Amazon EC2 instance and install Docker Engine and the Docker CLI.

**To install Docker on an Amazon EC2 instance**

Launch an instance with the Amazon Linux 2 or Amazon Linux AMI. For more information, see Launching an instance in the Amazon EC2 User Guide for Linux Instances.

Connect to your instance. For more information, see Connect to your Linux instance in the Amazon EC2 User Guide for Linux Instances.

Update the installed packages and package cache on your instance.

sudo yum update -y

Install the most recent Docker Engine package.

Amazon Linux 2

sudo amazon-linux-extras install docker

Amazon Linux.

sudo yum install docker

Start the Docker service.

sudo service docker start

(Optional) On Amazon Linux 2, to ensure that the Docker daemon starts after each system reboot, run the following command:

sudo systemctl enable docker

Add the ec2-user to the docker group so you can execute Docker commands without using sudo.

sudo usermod -a -G docker ec2-user

Log out and log back in again to pick up the new docker group permissions. You can accomplish this by closing your current SSH terminal window and reconnecting to your instance in a new one. Your new SSH session will have the appropriate docker group permissions.

Verify that the ec2-user can run Docker commands without sudo.

docker info

Note

In some cases, you may need to reboot your instance to provide permissions for the ec2-user to access the Docker daemon. Try rebooting your instance if you see the following error:

**Create a Docker image**

Amazon ECS task definitions use Docker images to launch containers on the container instances in your clusters. In this section, you create a Docker image of a simple web application, and test it on your local system or Amazon EC2 instance, and then push the image to a container registry (such as Amazon ECR or Docker Hub) so you can use it in an Amazon ECS task definition.

**To create a Docker image of a simple web application**

Create a file called Dockerfile. A Dockerfile is a manifest that describes the base image to use for your Docker image and what you want installed and running on it. For more information about Dockerfiles, go to the Dockerfile Reference.

touch Dockerfile

Edit the Dockerfile you just created and add the following content.

FROM ubuntu:18.04

# Install dependencies

RUN apt-get update && \

apt-get -y install apache2

# Install apache and write hello world message

RUN echo 'Hello World!' > /var/www/html/index.html

# Configure apache

RUN echo '. /etc/apache2/envvars' > /root/run\_apache.sh && \

echo 'mkdir -p /var/run/apache2' >> /root/run\_apache.sh && \

echo 'mkdir -p /var/lock/apache2' >> /root/run\_apache.sh && \

echo '/usr/sbin/apache2 -D FOREGROUND' >> /root/run\_apache.sh && \

chmod 755 /root/run\_apache.sh

EXPOSE 80

CMD /root/run\_apache.sh

This Dockerfile uses the Ubuntu 18.04 image. The RUN instructions update the package caches, install some software packages for the web server, and then write the "Hello World!" content to the web server's document root. The EXPOSE instruction exposes port 80 on the container, and the CMD instruction starts the web server.

**Build the Docker image from your Dockerfile.**

Note

Some versions of Docker may require the full path to your Dockerfile in the following command, instead of the relative path shown below.

docker build -t hello-world .

Run docker images to verify that the image was created correctly.

docker images --filter reference=hello-world

Output:

REPOSITORY TAG IMAGE ID CREATED SIZE

hello-world latest e9ffedc8c286 4 minutes ago 241MB

Run the newly built image. The -p 80:80 option maps the exposed port 80 on the container to port 80 on the host system. For more information about docker run, go to the Docker run reference.

docker run -t -i -p 80:80 hello-world

Note

Output from the Apache web server is displayed in the terminal window. You can ignore the "Could not reliably determine the server's fully qualified domain name" message.

Open a browser and point to the server that is running Docker and hosting your container.

If you are using an EC2 instance, this is the Public DNS value for the server, which is the same address you use to connect to the instance with SSH. Make sure that the security group for your instance allows inbound traffic on port 80.

If you are running Docker locally, point your browser to http://localhost/.

If you are using docker-machine on a Windows or Mac computer, find the IP address of the VirtualBox VM that is hosting Docker with the docker-machine ip command, substituting machine-name with the name of the docker machine you are using.

docker-machine ip machine-name

You should see a web page with your "Hello World!" statement.

Stop the Docker container by typing Ctrl + c.

Push your image to Amazon Elastic Container Registry

Amazon ECR is a managed AWS Docker registry service. You can use the Docker CLI to push, pull, and manage images in your Amazon ECR repositories. For Amazon ECR product details, featured customer case studies, and FAQs, see the Amazon Elastic Container Registry product detail pages.

This section requires the following:

You have the AWS CLI installed and configured. If you do not have the AWS CLI installed on your system, see Installing the AWS Command Line Interface in the AWS Command Line Interface User Guide.

Your user has the required IAM permissions to access the Amazon ECR service. For more information, see Amazon ECR managed policies.

To tag your image and push it to Amazon ECR

**Create an Amazon ECR repository to store your hello-world image.** Note the repositoryUri in the output.

Substitute region, with your AWS Region, for example, us-east-1.

aws ecr create-repository --repository-name hello-repository --region region

Output:

{

"repository": {

"registryId": "aws\_account\_id",

"repositoryName": "hello-repository",

"repositoryArn": "arn:aws:ecr:region:aws\_account\_id:repository/hello-repository",

"createdAt": 1505337806.0,

"repositoryUri": "aws\_account\_id.dkr.ecr.region.amazonaws.com/hello-repository"

}

}

Tag the hello-world image with the repositoryUri value from the previous step.

docker tag hello-world aws\_account\_id.dkr.ecr.region.amazonaws.com/hello-repository

Run the aws ecr get-login-password command. Specify the registry URI you want to authenticate to. For more information, see Registry Authentication in the Amazon Elastic Container Registry User Guide.

aws ecr get-login-password | docker login --username AWS --password-stdin aws\_account\_id.dkr.ecr.region.amazonaws.com

Output:

Login Succeeded

Push the image to Amazon ECR with the repositoryUri value from the earlier step.

docker push aws\_account\_id.dkr.ecr.region.amazonaws.com/hello-repository

Clean up

When you are done experimenting with your Amazon ECR image, you can delete the repository so you are not charged for image storage.

aws ecr delete-repository --repository-name hello-repository

**DOCKER**

**The Docker CLI**

Manage images

docker build

docker build [options] .

-t "app/container\_name" # name

Create an image from a Dockerfile.

docker run

docker run [options] IMAGE

# see `docker create` for options

Run a command in an image.

Manage containers

docker create

docker create [options] IMAGE

-a, --attach # attach stdout/err

-i, --interactive # attach stdin (interactive)

-t, --tty # pseudo-tty

--name NAME # name your image

-p, --publish 5000:5000 # port map

--expose 5432 # expose a port to linked containers

-P, --publish-all # publish all ports

--link container:alias # linking

-v, --volume `pwd`:/app # mount (absolute paths needed)

-e, --env NAME=hello # env vars

Example

$ docker create --name app\_redis\_1 \

--expose 6379 \

redis:3.0.2

Create a container from an image.

docker exec

docker exec [options] CONTAINER COMMAND

-d, --detach # run in background

-i, --interactive # stdin

-t, --tty # interactive

Example

$ docker exec app\_web\_1 tail logs/development.log

$ docker exec -t -i app\_web\_1 rails c

Run commands in a container.

docker start

docker start [options] CONTAINER

-a, --attach # attach stdout/err

-i, --interactive # attach stdin

docker stop [options] CONTAINER

Start/stop a container.

docker ps

$ docker ps

$ docker ps -a

$ docker kill $ID

Manage containers using ps/kill.

Images

docker images

$ docker images

REPOSITORY TAG ID

ubuntu 12.10 b750fe78269d

me/myapp latest 7b2431a8d968

$ docker images -a # also show intermediate

Manages images.

docker rmi

docker rmi b750fe78269d

Deletes images.

Also see

Getting Started (docker.io)

Dockerfile

Inheritance

FROM ruby:2.2.2

Variables

ENV APP\_HOME /myapp

RUN mkdir $APP\_HOME

Initialization

RUN bundle install

WORKDIR /myapp

VOLUME ["/data"]

# Specification for mount point

ADD file.xyz /file.xyz

COPY --chown=user:group host\_file.xyz /path/container\_file.xyz

Onbuild

ONBUILD RUN bundle install

# when used with another file

Commands

EXPOSE 5900

CMD ["bundle", "exec", "rails", "server"]

Entrypoint

ENTRYPOINT ["executable", "param1", "param2"]

ENTRYPOINT command param1 param2

Configures a container that will run as an executable.

ENTRYPOINT exec top -b

This will use shell processing to substitute shell variables, and will ignore any CMD or docker run command line arguments.

Metadata

LABEL version="1.0"

LABEL "com.example.vendor"="ACME Incorporated"

LABEL com.example.label-with-value="foo"

LABEL description="This text illustrates \

that label-values can span multiple lines."

See also

https://docs.docker.com/engine/reference/builder/

docker-compose

Basic example

# docker-compose.yml

version: '2'

services:

web:

build: .

# build from Dockerfile

context: ./Path

dockerfile: Dockerfile

ports:

- "5000:5000"

volumes:

- .:/code

redis:

image: redis

Commands

docker-compose start

docker-compose stop

docker-compose pause

docker-compose unpause

docker-compose ps

docker-compose up

docker-compose down

Reference

Building

web:

# build from Dockerfile

build: .

# build from custom Dockerfile

build:

context: ./dir

dockerfile: Dockerfile.dev

# build from image

image: ubuntu

image: ubuntu:14.04

image: tutum/influxdb

image: example-registry:4000/postgresql

image: a4bc65fd

Ports

ports:

- "3000"

- "8000:80" # guest:host

# expose ports to linked services (not to host)

expose: ["3000"]

Commands

# command to execute

command: bundle exec thin -p 3000

command: [bundle, exec, thin, -p, 3000]

# override the entrypoint

entrypoint: /app/start.sh

entrypoint: [php, -d, vendor/bin/phpunit]

Environment variables

# environment vars

environment:

RACK\_ENV: development

environment:

- RACK\_ENV=development

# environment vars from file

env\_file: .env

env\_file: [.env, .development.env]

Dependencies

# makes the `db` service available as the hostname `database`

# (implies depends\_on)

links:

- db:database

- redis

# make sure `db` is alive before starting

depends\_on:

- db

Other options

# make this service extend another

extends:

file: common.yml # optional

service: webapp

volumes:

- /var/lib/mysql

- ./\_data:/var/lib/mysql

Advanced features

Labels

services:

web:

labels:

com.example.description: "Accounting web app"

DNS servers

services:

web:

dns: 8.8.8.8

dns:

- 8.8.8.8

- 8.8.4.4

Devices

services:

web:

devices:

- "/dev/ttyUSB0:/dev/ttyUSB0"

External links

services:

web:

external\_links:

- redis\_1

- project\_db\_1:mysql

Hosts

services:

web:

extra\_hosts:

- "somehost:192.168.1.100"

sevices

To view list of all the services runnning in swarm

docker service ls

**To see all running services**

docker stack services stack\_name

to see all services logs

docker service logs stack\_name service\_name

To scale services quickly across qualified node

docker service scale stack\_name\_service\_name=replicas

clean up

To clean or prune unused (dangling) images

docker image prune

To remove all images which are not in use containers , add - a

docker image prune -a

To prune your entire system

docker system prune

To leave swarm

docker swarm leave

To remove swarm ( deletes all volume data and database info)

docker stack rm stack\_name

To kill all running containers

docker kill $(docekr ps -q )

**Amazon AppStream 2.0**

Amazon AppStream 2.0 uses EC2 instances to stream applications. You launch instances from base images, called image builders, which AppStream 2.0 provides. To create your own custom image, you connect to an image builder instance, install and configure your applications for streaming, and then create your image by creating a snapshot of the image builder instance.

**When you launch an image builder, you choose:**

An instance type — AppStream 2.0 provides different instance types with various compute, memory, and graphics configurations. The instance type must align with the instance family you need. For more information, see AppStream 2.0 Instance Families.

An operating system — AppStream 2.0 provides the following operating systems:

Windows Server 2012 R2

Windows Server 2016 Base

Windows Server 2019 Base

Amazon Linux 2

The subnet and security groups to use — Make sure that the subnet and security groups provide access to the network resources that your applications require. Typical network resources required by applications may include licensing servers, database servers, file servers, and application servers.

Fleets and Stacks

Fleet Type

The fleet type allows you to decide when your instances run and how you pay for them. When your instances runs determines how quickly your users application will launch when it is selected. You specify the fleet type when you create a fleet, and can't change the fleet type after it is created.

**The available fleet types are:**

**Always-On**

Streaming instances run continuously, even when no users are streaming applications and desktops.

**On-Demand**

Streaming instances run only when users are streaming applications and desktops. Streaming instances not yet assigned to users are in a stopped state.

**Elastic**

The pool of streaming instances is managed by AppStream 2.0. When your users select their application or desktop to launch, they will start streaming after the app block has been downloaded and mounted to a streaming instance. For more information about creating app blocks for your Elastic fleets, see Create and Manage App Blocks and Applications for Elastic Fleets.

Use an Always-On fleet to provide your users with instant access to their applications. Use an On-Demand fleet to optimize your streaming charges and provide your users with access to their applications after a 1-2 minute wait. For more information, see Amazon AppStream 2.0 Pricing.

Session Context

You can pass parameters to your streaming application by using either of the following methods:

Specify session content in the CreateStreamingURL AppStream 2.0 API operation. For more information, see CreateStreamingURL.

Specify the session context as a SAML assertion in your SAML 2.0 identity provider's authentication response. For more information, see Step 5: Create Assertions for the SAML Authentication Response.

If your image uses a version of the AppStream 2.0 agent that was released on or after October 30, 2018, the session context is stored within the image as a Windows or Linux environment variable. For information about specific environment variables, see "User and Instance Metadata for AppStream 2.0 Fleets" in Customize an AppStream 2.0 Fleet to Optimize Your Users' Application Streaming Experience .

Note

The session context parameter is visible to the user in the AppStream 2.0 streaming URL. We strongly recommend that you never put confidential or sensitive information in the session context parameter. Because it is possible for users to modify the streaming URL, we recommend performing additional validation to determine that the session context is valid for the end user. For example, you can compare the session context with other session information, such as user and instance metadata for AppStream 2.0 fleets.

AppStream 2.0 does not perform validation on the session context parameter.

Using Session Context to Pass Parameters to a Streaming Application

In the following steps, you'll use session context to start a web browser and automatically open a specific website. For instances running Windows, you'll use Firefox. For instances running Linux, you'll use Chromium.

To use session context to launch a website

In the left navigation pane, choose Images, Image Builder.

Choose the image builder to use, verify that it is in the Running state, and choose Connect.

Log in to the image builder by choosing Administrator on the Local User tab.

**Create a child folder of C:\.** For this example, use C:\Scripts.

**Create a Windows batch file in the new folder**. For this example, create C:\Scripts\session-context-test.bat and add a script that launches Firefox with the URL from the session context.

Use the following script:

CD "C:\Program Files (x86)\Mozilla Firefox"

Start firefox.exe %APPSTREAM\_SESSION\_CONTEXT%

In Image Assistant, add session-context-test.bat and change the name to Firefox.

You do not need to add Firefox. This step requires that you add only the batch file.

Create an image, fleet, and stack. For this example, use a fleet name of session-context-test-fleet and a stack name of session-context-test-stack.

After the fleet is running, you can call create-streaming-url with the session-context parameter, as shown in this example.

aws appstream create-streaming-url --stack-name session-context-test-stack \

--fleet-name session-context-test-fleet \

--user-id username –-validity 10000 \

--application-id firefox --session-context "www.amazon.com"

Open the streaming URL in a browser. The script file launches Firefox and loads http://www.amazon.com.

**Similarly, you can perform the following steps to pass parameters to your Linux streaming application.**

To pass parameters to your Linux streaming application

In the left navigation pane, choose Images, Image Builder.

Choose the image builder to use, verify that it is in the Running state, and choose Connect.

Log in to the image builder by default as ImageBuilderAdmin.

Create a script file (for example, launch-chromium.sh) by running the following command:

sudo vim /usr/bin/launch-chromium.sh

Write the script and set executable permissions, such as the following:

Note

#!/bin/bash and source /etc/profile are always required in the script.

#!/bin/bash

source /etc/profile

/usr/bin/chromium-browser $APPSTREAM\_SESSION\_CONTEXT

Use the Image Assistant CLI to add launch-chromium.sh:

sudo AppStreamImageAssistant add-application \

--name chromium \

--absolute-app-path /usr/bin/launch-chromium.sh

Create an image, fleet, and stack. For this example, use a fleet name of session-context-test-fleet and a stack name of session-context-test-stack.

After the fleet is running, you can call create-streaming-url with the session-context parameter, as shown in this example.

aws appstream create-streaming-url --stack-name session-context-test-stack \

--fleet-name session-context-test-fleet \

--user-id username \

--application-id chromium --session-context "www.amazon.com"

Open the streaming URL in a browser. The batch file launches Chromium and loads <http://www.amazon.com>.

Users can access AppStream 2.0 streaming sessions by using either a web browser or the AppStream 2.0 client on a supported device.

Depending on your organizational requirements, you can enable user access to AppStream 2.0 streaming sessions by: Setting up identity federation using SAML 2.0, using an AppStream 2.0 user pool, or creating a streaming URL. Following are recommendations for choosing a connection method.

SAML 2.0: Use this connection method when you have an identity provider that manages your user accounts and supports SAML 2.0 federation.

Note

This connection method is required when your AppStream 2.0 fleet is joined to a Microsoft Active Directory domain.

AppStream 2.0 user pools: Use this connection method when:

You want to set up a Proof-of-Concept (POC) quickly before you configure your SAML 2.0-compliant identity provider.

You don't have a SAML 2.0-compliant identity provider.

You want to manage users directly within the AppStream 2.0 console.

**Streaming URL:** Use this connection method when you want to programmatically provide access to AppStream 2.0 by using temporary URLs. We recommend this connection method when you want to use your existing identity provider to provide programmatic access to AppStream 2.0

**DEPLOY A CONTAINER TO AMAZON ECS( Github Workflow)**

This workflow will build and push a new container image to Amazon ECR, and then will deploy a new task definition to Amazon ECS, when there is a push to the main branch.

To use this workflow, you will need to complete the following set-up steps:

1. **Create an ECR repository to store your images.** For example: aws ecr create-repository --repository-name my-ecr-repo --region us-east-2. Replace the value of the ECR\_REPOSITORY environment variable in the workflow below with your repository's name. Replace the value of the AWS\_REGION environment variable in the workflow below with your repository's region.

**2. Create an ECS task definition, an ECS cluster, and an ECS service.**

For example, follow the Getting Started guide on the ECS console:

https://us-east-2.console.aws.amazon.com/ecs/home?region=us-east-2#/firstRun

Replace the value of the `ECS\_SERVICE` environment variable in the workflow below with the name you set for the Amazon ECS service.

Replace the value of the ECS\_CLUSTER environment variable in the workflow below with the name you set for the cluster.

**3. Store your ECS task definition as a JSON file in your repository.**

The format should follow the output of `aws ecs register-task-definition --generate-cli-skeleton`.

Replace the value of the `ECS\_TASK\_DEFINITION` environment variable in the workflow below with the path to the JSON file.

the value of the CONTAINER\_NAME environment variable in the workflow below with the name of the container in the containerDefinitions section of the task definition.

**4. Store an IAM user access key in GitHub Actions secrets** named AWS\_ACCESS\_KEY\_ID and AWS\_SECRET\_ACCESS\_KEY.

See the documentation for each action used below for the recommended IAM policies for this IAM user,

and best practices on handling the access key credentials.

name: Deploy to Amazon ECS

on: push: branches: - main

env: AWS\_REGION: ap-southeast-1 # set this to your preferred AWS region, e.g. us-west-1 ECR\_REPOSITORY: classificationmodel # set this to your Amazon ECR repository name ECS\_SERVICE: classifcationmodelservice # set this to your Amazon ECS service name ECS\_CLUSTER: classificationmodelcluster # set this to your Amazon ECS cluster name ECS\_TASK\_DEFINITION: classificationmodeltask.json# set this to the path to your Amazon ECS task definition # file, e.g. .aws/task-definition.json CONTAINER\_NAMEclassificationmodelcontainer # set this to the name of the container in the # containerDefinitions section of your task definition

jobs: deploy: name: Deploy runs-on: ubuntu-latest environment: production

steps:

- name: Checkout

uses: actions/checkout@v2

- name: Configure AWS credentials

uses: aws-actions/configure-aws-credentials@v1

with:

aws-access-key-id: ${{ secrets.AWS\_ACCESS\_KEY\_ID }}

aws-secret-access-key: ${{ secrets.AWS\_SECRET\_ACCESS\_KEY }}

aws-region: ${{ env.AWS\_REGION }}

- name: Login to Amazon ECR

id: login-ecr

uses: aws-actions/amazon-ecr-login@v1

- name: Build, tag, and push image to Amazon ECR

id: build-image

env:

ECR\_REGISTRY: ${{ steps.login-ecr.outputs.registry }}

IMAGE\_TAG: ${{ github.sha }}

run: |

# Build a docker container and

# push it to ECR so that it can

# be deployed to ECS.

docker build -t $ECR\_REGISTRY/$ECR\_REPOSITORY:$IMAGE\_TAG .

docker push $ECR\_REGISTRY/$ECR\_REPOSITORY:$IMAGE\_TAG

echo "::set-output name=image::$ECR\_REGISTRY/$ECR\_REPOSITORY:$IMAGE\_TAG"

- name: Fill in the new image ID in the Amazon ECS task definition

id: task-def

uses: aws-actions/amazon-ecs-render-task-definition@v1

with:

task-definition: ${{ env.ECS\_TASK\_DEFINITION }}

container-name: ${{ env.CONTAINER\_NAME }}

image: ${{ steps.build-image.outputs.image }}

- name: Deploy Amazon ECS task definition

uses: aws-actions/amazon-ecs-deploy-task-definition@v1

with:

task-definition: ${{ steps.task-def.outputs.task-definition }}

service: ${{ env.ECS\_SERVICE }}

cluster: ${{ env.ECS\_CLUSTER }}

wait-for-service-stability: true

**COMMANDS TO DEPLOY ELK IN EC2**

ssh -i <path to .pem file> ubuntu@<dns name>

sudo apt-get update

sudo apt-get upgrade

sudo apt-get install default-jre

java -version

sudo apt-get install nginx

sudo systemctl start nginx

sudo systemctl status nginx

wget -qO - https://artifacts.elastic.co/GPG-KEY-elasticsearch | sudo apt-key add -

echo "deb https://artifacts.elastic.co/packages/6.x/apt stable main" | sudo tee -a /etc/apt/sources.list.d/elastic-6.x.list

sudo apt-get update

sudo apt-get install elasticsearch

sudo nano /etc/elasticsearch/elasticsearch.yml

network.host: 0.0.0.0

http.cors.enabled: true

http.cors.allow-origin: "\*"

sudo nano /etc/elasticsearch/jvm.options

-Xms128m

-Xmx128m

sudo systemctl start elasticsearch

sudo systemctl enable elasticsearch

sudo systemctl status elasticsearch

sudo curl -XGET http://localhost:9200

sudo apt-get install kibana

sudo systemctl start kibana

sudo systemctl enable kibana

sudo systemctl status kibana

echo "kibanaadmin:`openssl passwd -apr1`" | sudo tee -a /etc/nginx/htpasswd.users

sudo nano /etc/nginx/sites-available/13.215.36.2

server {

listen 80;

server\_name 13.215.36.2;

auth\_basic "Restricted Access";

auth\_basic\_user\_file /etc/nginx/htpasswd.users;

location / {

proxy\_pass http://localhost:5601;

proxy\_http\_version 1.1;

proxy\_set\_header Upgrade $http\_upgrade;

proxy\_set\_header Connection 'upgrade';

proxy\_set\_header Host $host;

proxy\_cache\_bypass $http\_upgrade;

}

}

sudo ln -s /etc/nginx/sites-available/13.215.36.2 /etc/nginx/sites-enabled/13.215.36.2

sudo nginx -t

sudo systemctl restart nginx

sudo ufw allow 'Nginx Full'

scp -i <path to .pem file> <path to sample-movies.bulk file> ubuntu@<dns name>:sample-movies.bulk

sudo curl -XPOST http://localhost:9200/\_bulk?pretty --data-binary @sample-movies.bulk -H "Content-Type: application/json"