DevSecOps Training

https://git.rockfin.com/training-iac/training-starter-kit

https://git.rockfin.com/training-iac/training-starter-kit/blob/master/README.md

Training recordings:

Day 1:
https://rockfin.sharepoint.com/sites/DevOpsTrainingClass/Shared%20Documents/Main%20Channel/Recordings/Meeting%20now-20210621_131605-Meeting%20Recording.mp4?web=1

Day 2:
Day 2 – part 1:
https://rockfin.sharepoint.com/sites/DevOpsTrainingClass/Shared%20Documents/Main%20Channel/Recordings/Meeting%20in%20_Main%20Channel_-20210622_100642-Meeting%20Recording.mp4?web=1

Day 2 – Part 2:
https://rockfin.sharepoint.com/sites/DevOpsTrainingClass/Shared%20Documents/Main%20Channel/Recordings/Meeting%20in%20_Main%20Channel_-20210622_132712-Meeting%20Recording.mp4?web=1
https://rockfin.sharepoint.com/sites/DevOpsTrainingClass/Shared%20Documents/Main%20Channel/Recordings/Meeting%20in%20_Main%20Channel_-20210622_100642-Meeting%20Recording.mp4?web=1

Day 3:
Day 3 – part 1:
https://rockfin.sharepoint.com/sites/DevOpsTrainingClass/Shared%20Documents/Main%20Channel/Recordings/Meeting%20in%20_Main%20Channel_-20210623_101439-Meeting%20Recording%201.mp4?web=1

Day 3 – part 2:
https://rockfin.sharepoint.com/sites/DevOpsTrainingClass/Shared%20Documents/Main%20Channel/Recordings/Meeting%20in%20_Main%20Channel_-20210623_101439-Meeting%20Recording.mp4?web=1

training-iac/training-starter-kit

-

Training Workshop Kit

Docker : Open platform for developers and sysadmins to build, ship, and run distributed applications as containers, whether on laptops, data center VMs, or the cloud.
CircleCI : Continuous Integration (CI) and Continuous Deployments (CD) tool
AWS Elastic Container Service (ECS) : Orchestration of Docker containers on EC2 instances. This runs the code, auto-scales containers and the EC2's they run on up and down, as well as handling the infrastructure side of deploying new code.
HAL9000 : QL in-house application that handles deployments and releases to various environments (including AWS). It also handles a lot of the encrypted configuration values which are passed to the code as Environment Variables.
Terraform : Terraform is Infrastructure-As-Code (IAC) that can build out highly QL-specific infrastructure components in a repeatable way. Instead of a lot of click-and-configure, we can run terraform scripts to create approved infrastructure stacks on-demand in minutes.

Prerequisites

These are the prereqs which are needed prior to coming to a workshop. The following should be completed and ready to go:

- Docker installed locally (**Install through Software Center (PC) or Self Service (Mac)** - If you install from the Docker web site, you will be missing key network policies needed)
    - If using a Windows PC, make sure right-click the Docker icon in your system tray and make sure it's set to use LINUX containers. In other words, it should set "Switch to Windows containers", which means it's currently on Linux containers.
    - Once the above steps are complete (regardless of operating system), open a command prompt/terminal window, enter the following command, and wait for it to complete: "docker pull microsoft/dotnet:2.1-sdk". **Verify this steps works prior to the class**
- VSCode or similar file editor (https://code.visualstudio.com/download)
- AWSCLI installed (https://docs.aws.amazon.com/cli/latest/userguide/install-cliv2.html)
    - Go to a command prompt, type in **aws configure**. Hit enter for the first 2 lines, enter "us-east-2" for the region, then enter again. This will create an **.aws** folder under your user folder with configuration files for the cli.
    - Due to our network firewall, you need to add the QL certs into the certificate chain so the awscli can properly communicate with AWS. Go here: https://git.rockfin.com/raw/sudoers/ssm-instance-connect/master/files/qlcerts.pem, download that file to your **~/.aws/** folder. Add the following "ca_bundle" line to your ~/.aws/config file:

```
[default]
region = us-east-2
ca_bundle = ~/.aws/qlcerts.pem
```

- Request AWS account access to Training Lab Account (418023852230). Navigate to myaccess/ in your browser, click "Add/Remove Access", and search for `AWS-SSO-TrainingIAC-NonProd-ServerEngineer`. Click the checkbox and request it for your **dash account**. Then click "Review and Submit", then "Submit".
- Terraform (https://www.terraform.io/)
    - If using a Windows PC, wherever you download Terraform to, make sure to add that path to your PATH variable. See here for how: https://helpdeskgeek.com/windows-10/add-windows-path-environment-variable/
    - If Mac, see here: https://osxdaily.com/2014/08/14/add-new-path-to-path-command-line/
    - Effectively, you want to be in any folder on your computer and be able to call that binary. To test, go to the command line of your OS and type "terraform version": you should get the version back. If not, see someone from IT Team Sudo.
- Create demo repository under your personal QL GIT org (https://git.rockfin.com/YOURNAME/999999-YOURNAME-iac)
- If you've never used HAL before, go to https://hal.zone/ in your browser and login with your dash account. This will create your account in HAL.
- Clone example GIT project locally (see below)

Setting up GIT

1. Clone this repo locally to your laptop (https://git.rockfin.com/training-iac/training-starter-kit.git)

2. Rename the folder to this naming standard: **999999-YOURNAME-iac**, where YOURNAME is your AD network login (first initial, last name). Then, delete the **.git** folder at the root; this allows you to associate this folder with a different git repo. This will become the base for the Git repo for your application's infrastructure.

3. Make a repo with the same name as what you called the folder in step 2 under your personal Git organization. https://git.rockfin.com/YOUR-NAME/999999-YOURNAME-iac. For example, https://git.rockfin.com/jsmith/999999-jsmith-iac

4. Follow the instructions given to you post-creation in your browser to upload your modified clone of the starter-kit to your applications new IAC repo. The one difference is instead of "git add README.md", type "git add -A" instead to add everything.

You are all set if you have a custom-named version of the starter kit uploaded and ready to go in your personal (QL-owned) GIT org.

:Commands:

Docker help command: docker <command> --help : eg: docker rmi --help

Docker run in Detach mode: docker run -d -p 5000:80 ppabbisetty

Docker stop: docker stop <first 3 letters of running container name> : eg: docker stop 434

Docker process: docker ps

Docker list: docker ls

Docker images: docker images.

Docker prune images (eg: <none> tag images): docker images prune

Docker re-run upgrade images: (don't run on same port, if other docker is running): docker run -d -p 5000:80 ppabbisetty:2.0

Stopping docker: Docker stop <container id / first 3 letters of dockert>:

Docker ps : docker process

Docker stat: latest statistics: docker stats

Docker remove images: docker rmi <image name>: docker rmi 66cb81dbcea8

Docker command line: https://docs.docker.com/compose/completion/

Command to navigate files within docker: docker exec -it <image id> bash.

Eg: docker exec -it d4439378e06f6d00c29e56f7083c0a4c5dc46af8b628952d280966575f916a03 bash

Auto generate docker configuration file from visual studio code:

1. Install docker component.
2. Navigate to command pallete : and search for 'Add docker compose files to workspace" and continue the steps to fill name, port, etc for generating the docker compose file.

DOCKERFILE BEST PRACTICES

- Minimize the number of layers/steps in the Dockerfile

Minimizing the number of steps in your image may improve build and pull performance

5 steps (layers)

```
FROM alpine:3.4

RUN apk update
RUN apk add curl
RUN apk add vim
RUN apk add git
```

2 steps (layers)

```
FROM alpine:3.4

RUN apk update && \
    apk add curl && \
    apk add vim && \
    apk add git
```

- Keep in mind that only RUN, COPY and ADD instructions create layers

DOCKERFILE BEST PRACTICES

- Start your Dockerfile with the steps that are least likely to change. This will use layers and caching to their fullest effect.

The best practice is to structure your Dockerfile according to the following:

1. Install tools that are needed to build your application.
2. Install dependencies, libraries and packages.
3. Build your application.

- Use a .dockerignore file

The directory where you issue the docker build command is called the build context. Docker will send all of the files and directories in your build directory to the Docker daemon as part of the build context. If you have stuff in your directory that is not needed by your build, you'll have an unnecessarily larger build context that results in a larger image size.

You can remedy this situation by adding a .dockerignore file that works similarly to .gitignore. You can specify the list of folders and files that should be ignored in the build context.

DOCKERFILE BEST PRACTICES

- Containers should be ephemeral

This would belong to generic Docker guidelines, but it's never enough to stress this point. It is your best interest to design and build Docker images that can be destroyed and recreated/replaced automatically or with minimal configuration.

- One container should have one concern

Think of containers as entities that take responsibility for one aspect of your project. So design your application in a way that your web server, database, in-memory cache and other components have their own dedicated containers.

DOCKERFILE INSTRUCTIONS BEST PRACTICES

- FROM - what image you are using. Can also alias images with multi-stage builds.
- COPY - copy files into the container
- ENV - sets environment variables
- RUN - run commands.
- WORKDIR - set the working directory.
- CMD/ENTRYPOINT - specifies command to run when container starts

More details in Training GIT:
https://git.rockfin.com/training-iac/training-starter-kit/blob/master/docs/Docker/DockerFile-Commands.docx

Voting app: https://github.com/dockersamples/example-voting-app

Tomorrow things need to have:

1. AWS CLI installed
2. Requested AWS access –
3. Pre-req : install all

Terraform folder: C:\Pavan\terraform

Day 2:

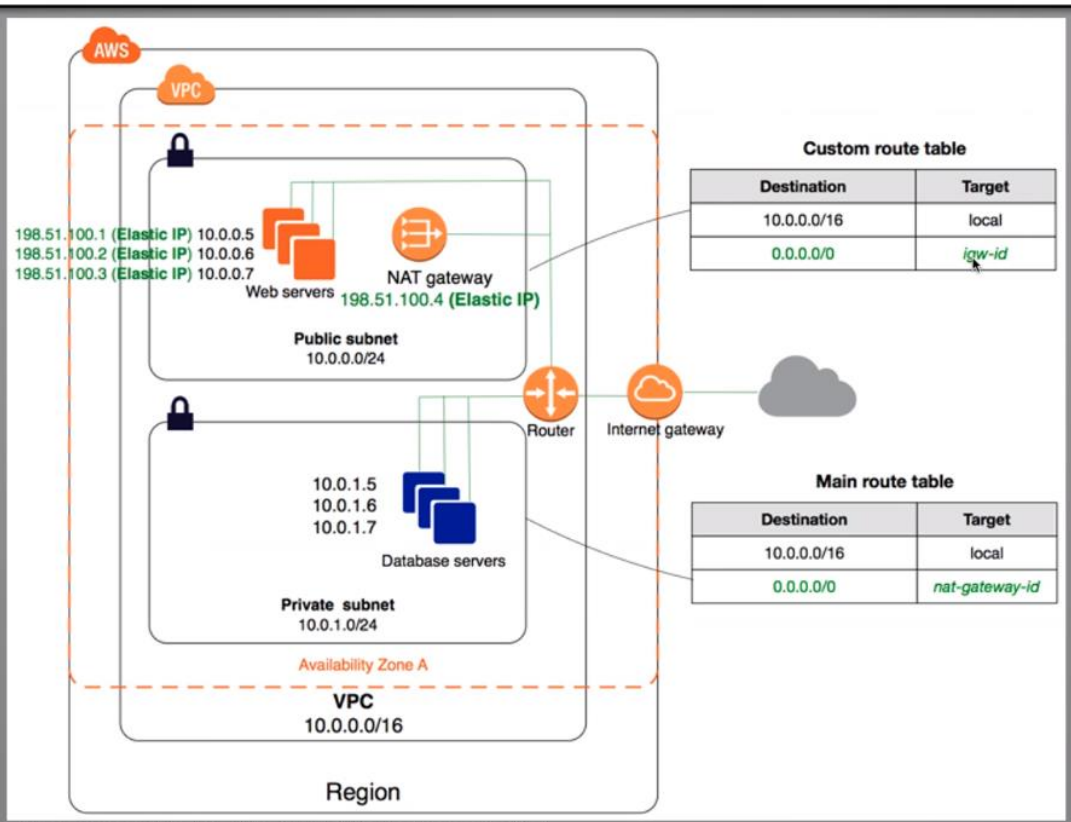
VPC
VIRTUAL PRIVATE CLOUD

- **VPC** – Virtual Private Cloud. AWS VPC lets you provision a logically isolated section of the AWS Cloud where you can launch AWS resources in a virtual network that you define. Effectively, your own datacenter!
- **Subnets** – Subnetworks of a VPC. Example is taking a pie which is the VPC and cutting it into smaller pieces. In our case, we use subnets to segment public, private, and persistent data resources.
- **NAT Gateway** – Translates private IP to a public IP. Your router at home as an example does this. Resources behind this are NOT publicly exposed to the internet.
- **IGW (internet Gateway)** – A VPC component that allows communication between instances in your VPC and the internet **directly**. Your modem would be what an IGW does for AWS as a comparison. Resources behind this ARE publicly exposed to the internet.
- **VPC Peering / VPN / DirectConnect** – Different ways to connect VPC's to other networks. VPC Peering allows VPC's to connect together, VPN / DirectConnect is used to connect back onprem.

Articles: https://docs.aws.amazon.com/vpc/latest/userguide/what-is-amazon-vpc.html

High availability – deploying application in EAST region with different locations like east-1, east-2, east-3.

VPC
VIRTUAL PRIVATE CLOUD



Articles: https://docs.aws.amazon.com/vpc/latest/userguide/what-is-amazon-vpc.html

IAM

IDENTITY AND ACCESS MANAGEMENT

- A web service that helps you control access to AWS resources. Consists of 3 types of authentication: Users, Group, and Roles.
- *A user* is an entity that you create in AWS to represent the person or application that uses it to interact with AWS. You can then allow either a.) access to the console by generating a password or b.) programmatic access to the API only by generating access and secret keys. Groups are simply grouping users together; we don't really use these at QL.
- Roles can be assigned to users (such as when you sign into awsconsole/ and log on as AWS-SSO-Developer) or compute resources (such as EC2, Lambda, or an ECS container).
- **Policies** are then attached to **roles** or **users** to grant **permissions**.
- Policies (what give actual permissions) are constructed as such:
  - Effect : Allow or Deny
  - Action: What specific action(s) are we allowing or denying
  - Resource: What specific resource(s) are we allowing or denying that action on
- A role's *trust relationship* tab shows what AWS service can use, or "assume" that role, as well as how long ago it actually did.

brycedev-ecscluster-418023852230-us-east-2-notifies-sns	**AWS service:** autoscaling	None
ecsAutoscaleRole	**AWS service:** application-autoscaling	None
ecsInstanceRole	**AWS service:** ec2	121 days

IAM access is given for the AWS resources which can't be in VPN like S3, ec2, lamada, etc.
Could check on policies, roles, user groups

IAM

IDENTITY AND ACCESS MANAGEMENT

- Managed policies are AWS provided and generally used to provide infrastructure boilerplate access to backend AWS services , for example EC2's having an ECS managed policy role on them so they can communicate with the cluster services.

AmazonEC2FullAccess	AWS managed	Permissions policy (3)	Provides full access to Amazon EC2 via the AWS Management Console.
AmazonEC2ReadOnlyAccess	AWS managed	None	Provides read only access to Amazon EC2 via the AWS Management Console.

- Inline policies are user-provided and give access to specific resources for use by your applications. They are scoped to only the role in which they were created; cannot be shared across multiple roles.

This is a sample policy giving Get / Describe DynamoDB access to the table "ratelock-api":

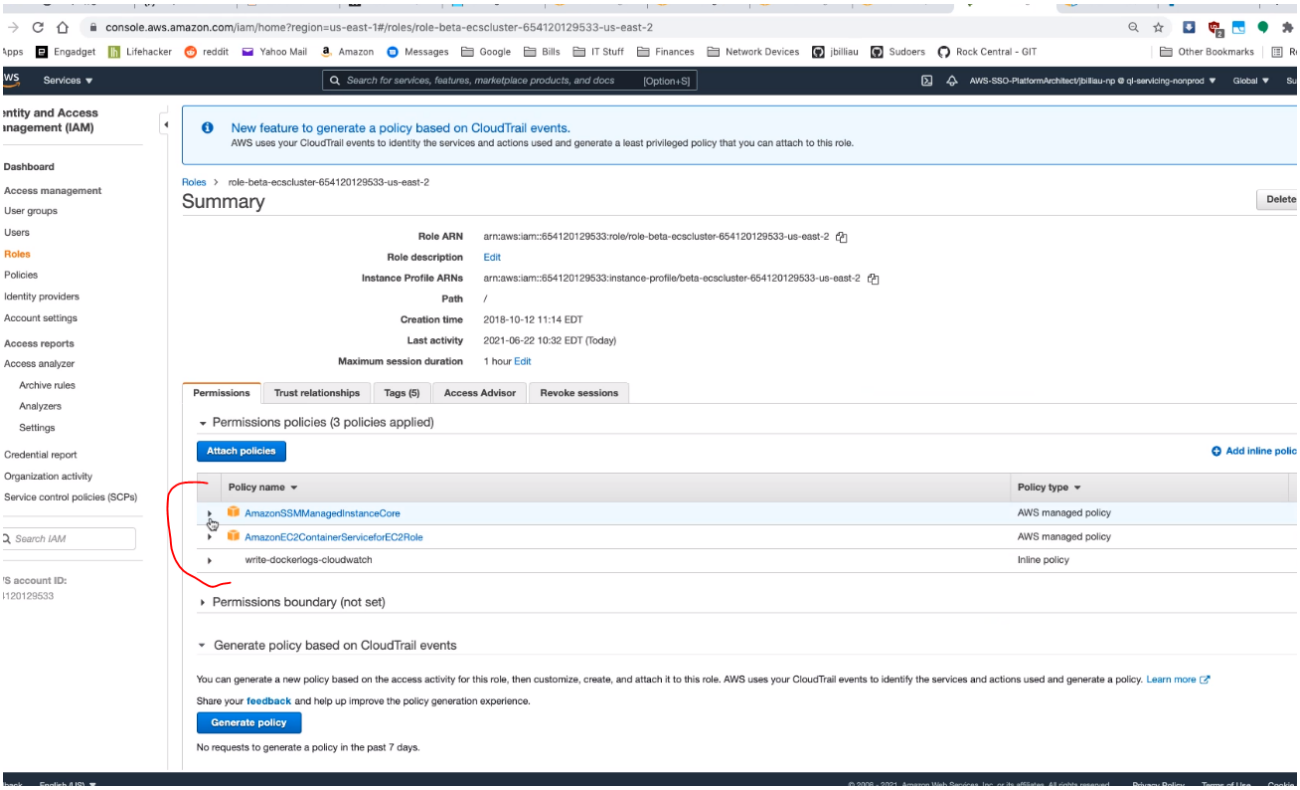
```
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Sid": "VisualEditor0",
      "Effect": "Allow",
      "Action": [
        "dynamodb:Describe*",
        "dynamodb:Get*"
      ],
      "Resource": "arn:aws:dynamodb:us-east-2:418023852230:table/ratelock-api"
    }
  ]
}
```

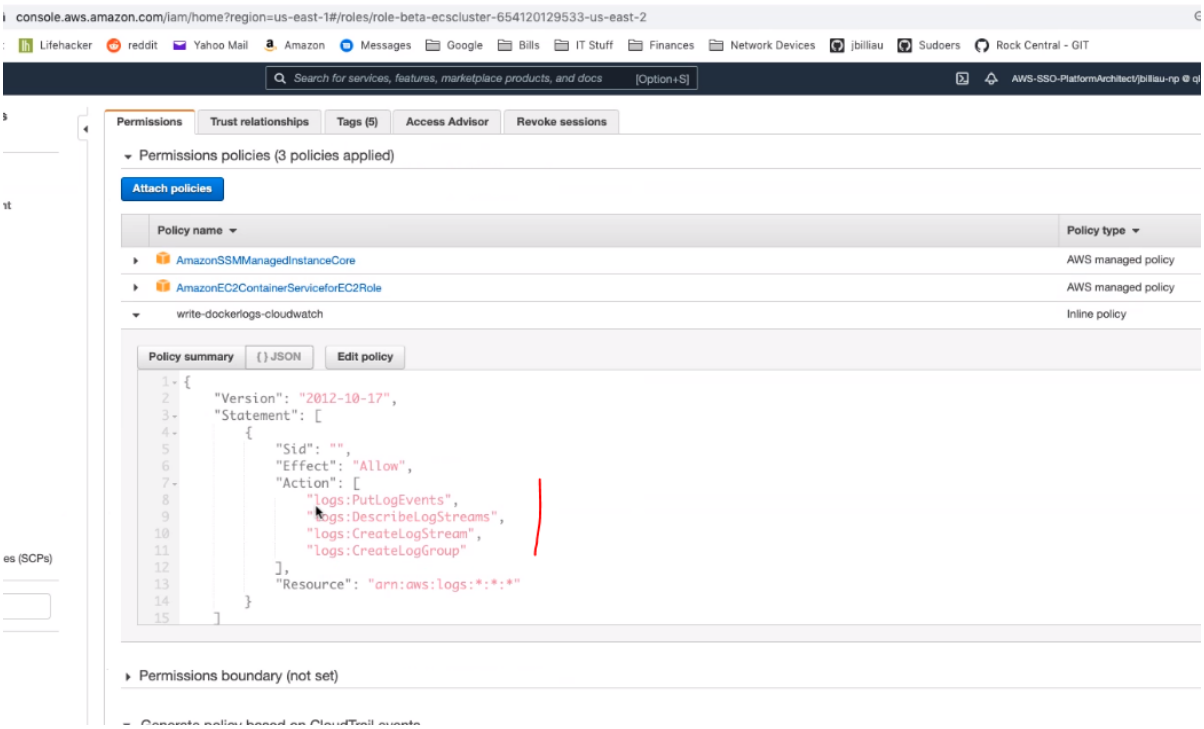
This is a sample policy giving PutItems S3 access to the bucket "documents":

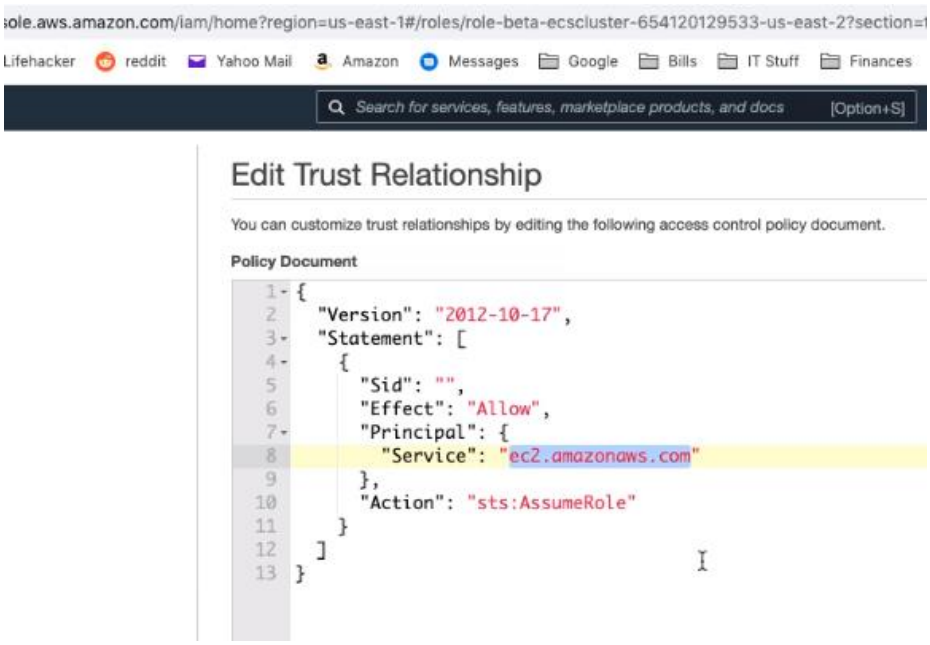
```
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Sid": "VisualEditor0",
      "Effect": "Allow",
      "Action": "s3:PutItem",
      "Resource": "arn:aws:s3:::documents"
    }
  ]
}
```

Article: https://docs.aws.amazon.com/IAM/latest/UserGuide/introduction.html

Most frequently used is ‘inline policies’ give specific permission to specific resources.







EC2: (Elastic Compute Cloud)

EC2
ELASTIC COMPUTE CLOUD

- VM's in the cloud known as *instances*
- Able to quickly deploy instances from AMI's *(Amazon Machine Image's)*
- Full control over your VM's – SSH / Remote Desktop
- Can be assigned security groups which act as a firewall
- Can be assigned IAM Roles allowing granular access to other AWS resources
- Can be scaled via auto-scaling groups and load balanced via load balancers.

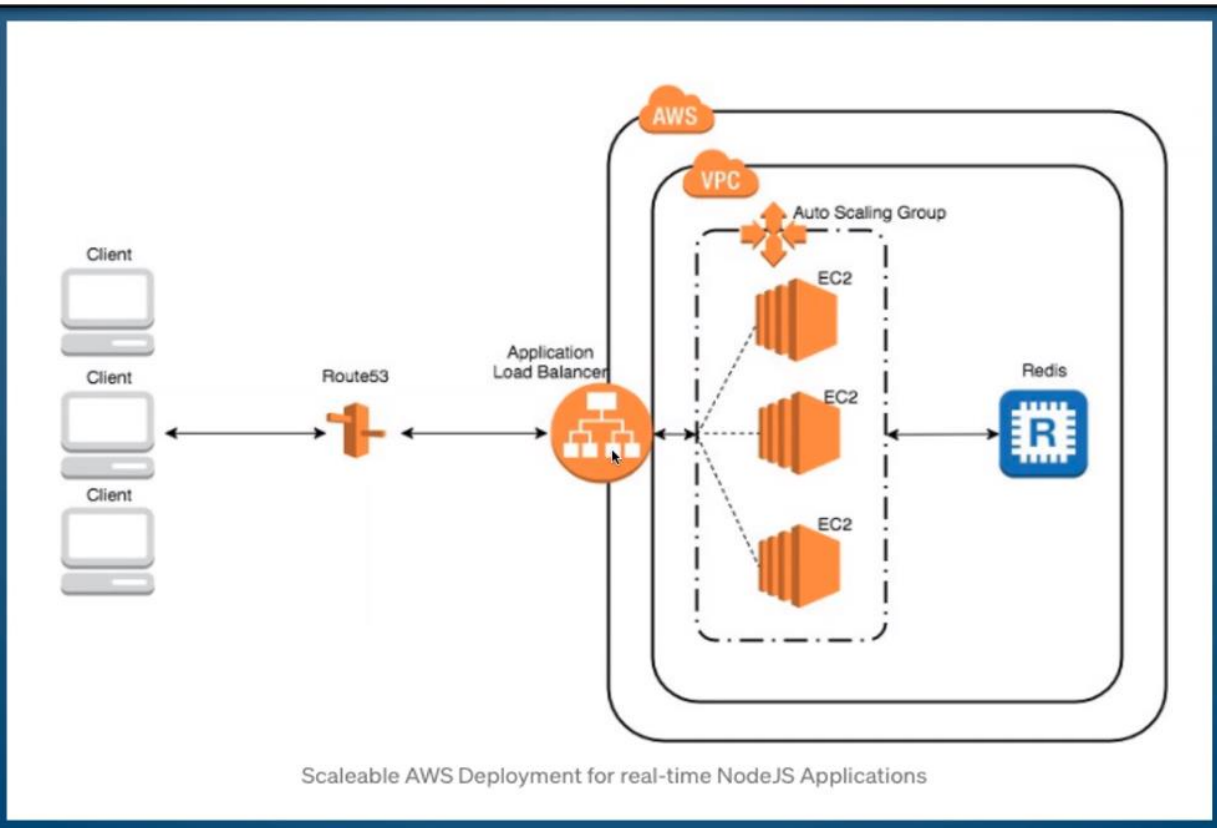
EC2
INSTANCE TYPES

	General Purpose		Compute Optimized	Memory Optimized		Accelerated Computing	Storage Optimized		
Type	t2	m5	c5	r4	x1e	p3	h1	i3	d2
Description	Burstable, good for changing workloads	Balanced, good for consistent workloads	High ratio of compute to memory	Good for in-memory databases	Good for full in-memory applications	Good for graphics processing and other GPU uses	HDD backed, balance of compute and memory	SDD backed, balance of compute and memory	Highest disk ratio
Mnemonic	**t** is for **tiny** or **turbo**	**m** is for **main** or happy **medium**	**c** is for **compute**	**r** is for **RAM**	**x** is for **xtreme**	**p** is for **pictures**	**h** is for **HDD**	**i** is for **IOPS**	**d** is for **dense**

Articles: https://docs.aws.amazon.com/AWSEC2/latest/UserGuide/concepts.html

EC2

ELASTIC COMPUTE CLOUD



Articles: https://docs.aws.amazon.com/AWSEC2/latest/UserGuide/concepts.html

Autoscaling and Load balancer:

Beanstalk

ELASTIC BEANSTALK

- AWS's PaaS (Platform-as-a-Service) – Easy way to get started in AWS and start deploying your app.
- Was used the most for QL's first foray into AWS; has since been replaced by ECS+Lambda.
- Features:
  - Capacity Provisioning (EC2)
  - Load Balancing (ELB)
  - Auto-Scaling (Auto-scaling Group + Launch Configuration)
  - Monitoring / Logging (Cloudwatch)
  - Version Control
  - Log file rotation to S3
- Does nothing you couldn't do yourself if you setup manually: is just an orchestrator of all of the features above.
- Supports blue/green deployments, multiple environments, cloning of environments.

:

ECS: Elastic Container Service

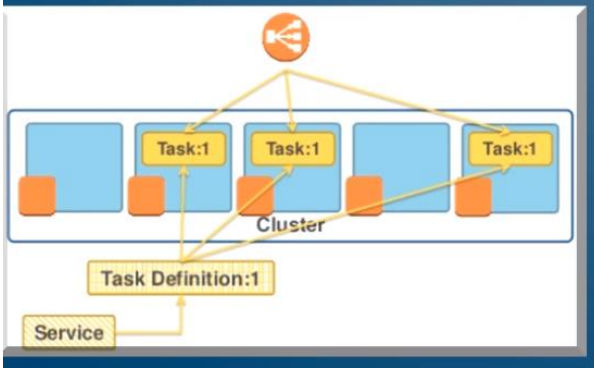
ECS
ELASTIC CONTAINER SERVICE

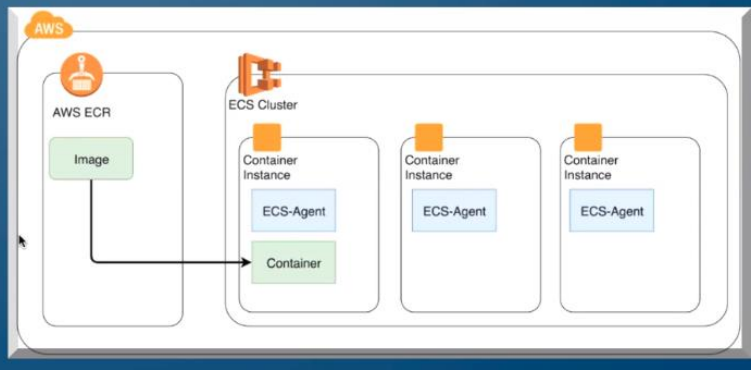
- Amazon ECS is a highly scalable, high-performance container orchestration service that supports Docker containers and allows you to easily run and scale containerized applications on AWS.
- You manage the EC2 instances, ECS takes care of the orchestration.
- AWS Fargate is an ECS-variant that takes it a step further and takes away all management of EC2's, allowing you to simply upload your container and AWS runs it for you.
- Fargate has some key difference between it and traditional ECS:
  - No access to underlying container, since it's running on an AWS EC2 you have no control over.
  - More expensive than ECS, since you are no longer achieving density of multiple containers per EC2.
  - Creates an ENI into your VPC, which allows for unique IP and security groups to be applied.
  - No ability to burst, limited to just the task size you specified; on the flip side, no "noisy neighbor" issues.
- Supports blue/green deployments, auto-scaling based off configured metrics/alarms, and IAM roles to be assigned.

Article: https://docs.aws.amazon.com/AmazonECS/latest/developerguide/Welcome.html

ECS (CONTINUED)
ELASTIC CONTAINER SERVICE

- Docker images are pushed to ECR (Elastic Container Registry)
- Task Definitions are the blueprint for your application: dictate what image is being used, CPU / RAM reservations, env variables, logging, etc.
- Tasks are instances of the container
- Services are orchestrations of tasks





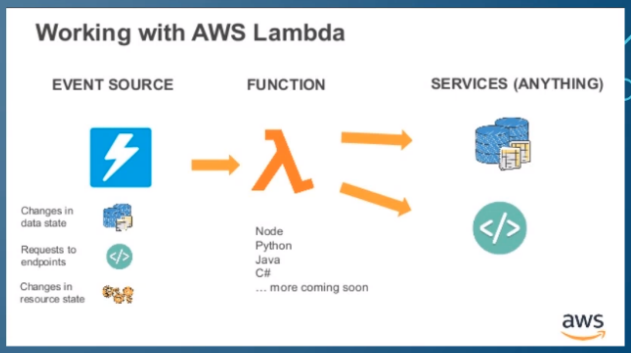
Cloudwatch monitoring service: shows what services/tasks and container running.

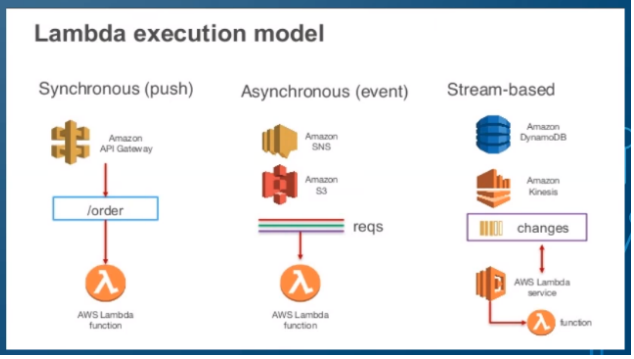
Lambda:

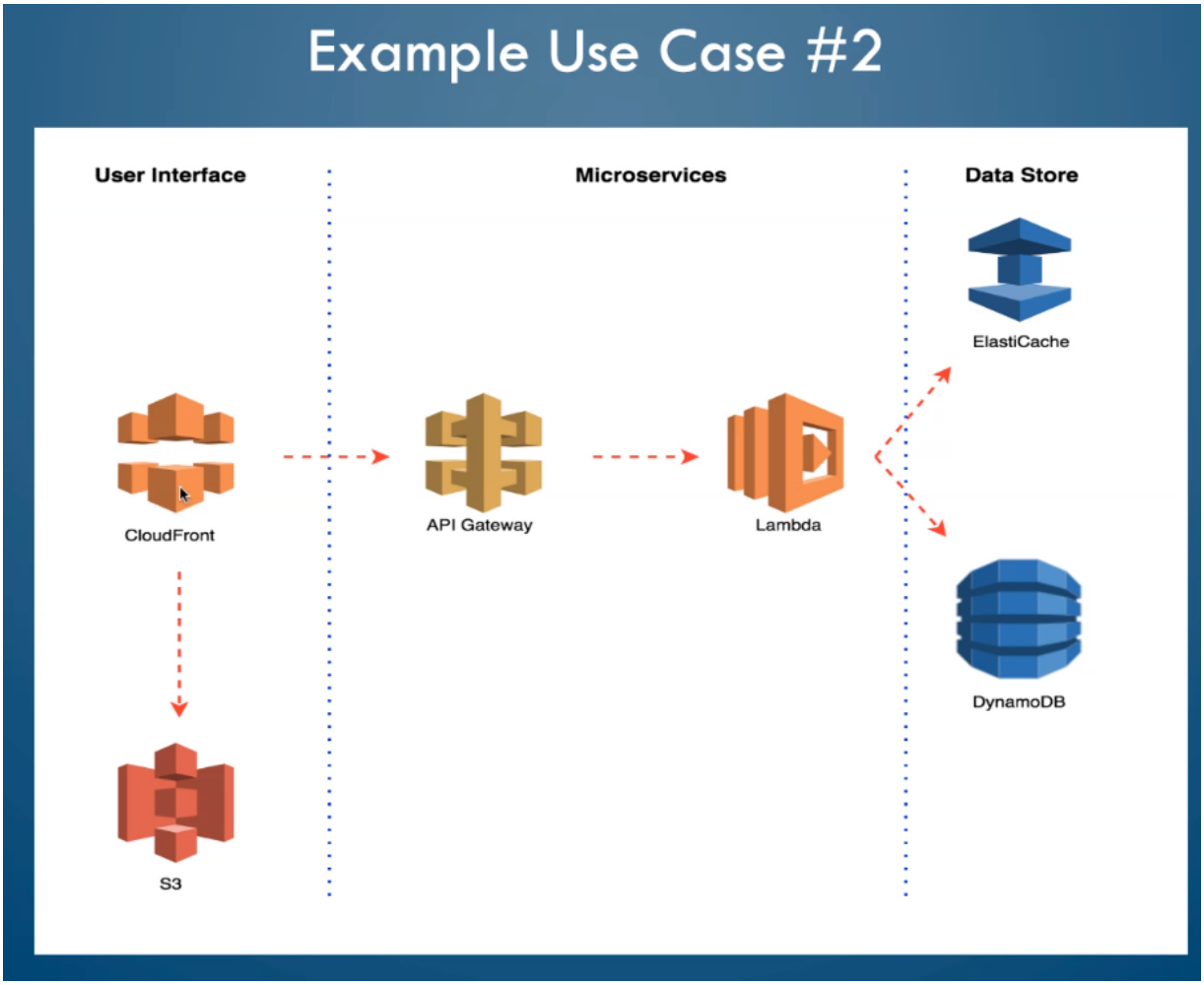
LAMBDA

- Serverless computing – upload code and go! Code can be uploaded via ZIP or edited inline in console.
- Languages available: Node.js, Java, C#, Go and Python, .Net CORE
- AWS Lambda executes your code only when needed and scales automatically, from a few requests per day to thousands per second. You pay only for the compute time you consume - there is no charge when your code is not running
- Configurable with up to 3 GB of RAM, 15 minute max runtime
- Can be invoked via SQS, SNS, API Gateway, Cloudwatch Events, and more

Article: https://docs.aws.amazon.com/lambda/latest/dg/welcome.html







Training practice:

https://git.rockfin.com/training-iac/training-starter-kit/tree/master/docs/AWS

ECS LAB

This will create an ECS service that runs your custom container in Fargate.

Upload your local docker image to Elastic Container Registry (ECR)

First, you will need to configure your AWS CLI with the correct credentials with access to push to ECR. If you use the **SAML TO STS** Google Chrome plugin, you already are all set with the credentials that are downloaded when you log into the training-iac account.

If you don't have that plugin and/or this is your first time using the AWS CLI, configure your CLI as below, with credentials you'll be given during the class:

```
$ aws configure

AWS Access Key ID: *GIVEN IN CLASS*
AWS Secret Access Key: *GIVEN IN CLASS*
Default region name: us-east-2
Default output format: skip, just hit enter
```

1. Type `aws --version` and see what version of the CLI you have installed. Depending if it's 1.XX or 2.XX, and if you have a Mac or PC, use ONE of the below 3 commands to login to ECR:

    i. **AWS CLI v2 (Windows/Mac)**

    ```
    aws ecr get-login-password --region us-east-2 --no-verify-ssl | docker login --username
    AWS --password-stdin 418023852230.dkr.ecr.us-east-2.amazonaws.com
    ```

    ii. **AWS CLI v1 (Windows Powershell)**

    ```
    Invoke-Expression -Command (aws ecr get-login --region us-east-2 --no-include-email)
    ```

    iii. **AWS CLI v1 (Mac)**

    ```
    eval $(aws ecr get-login --region us-east-2 --no-include-email)
    ```

2. If any of above commands fails with an invalid security token, open your credentials file in a text editor (probably C:\Users\yourname.aws\credentials) and delete the line starting with aws_session_token, then run the command again.

3. If AWS CLI v1, take outputted "docker login" command, copy and paste it, then run it

4. docker tag **YOURNAME:YOURVERSION** 418023852230.dkr.ecr.us-east-2.amazonaws.com/training-ecr:**YOURNAME**

5. docker push 418023852230.dkr.ecr.us-east-2.amazonaws.com/training-ecr:**YOURNAME**

At this point, you have taken your local Docker image you built in the Docker class, tagged it to AWS ECR, and pushed it there. Your Docker image is now in a cloud repository. Let's create a task definition and then ECS Service to deploy it!

Create ECS Task Definition

This is the blueprint for your application, the ECS equivalent of the docker compose file. It controls what image to run, how many resources to grant, environment variables, logging, etc.

1. Login to awsconsole and navigate to the ECS console. Click **Task Definitions**.
2. Click **Create new Task Definition**. On the next screen, click **FARGATE** and then next step.
3. For *Task Definition Name*, put in your first initial and last name.
4. For *Task Role*, select "None".
5. For *Task Execution Role*, make sure it's set to "ecsTaskExecutionRole*. For *Task Size*, select **1GB** for Task Memory, **.5 vCPU** for Task CPU.
6. Scroll down and click **Add Container**.
    - For *Container name*, put in your first initial, last name again.
    - for *Image*, put in the link to your ECR image you uploaded in step 4 above: 418023852230.dkr.ecr.us-east-2.amazonaws.com/training-ecr:**YOURNAME**
    - For *Memory limits*, put in **128**.
    - For *Port mappings*, put in **80** for the "container port".
    - Click **Add** in the bottom right.
    - Scroll all the way down and click **Create**. If successful, click **View Task Definition** to go back to the main console area.

Create ECS Service

Amazon ECS allows you to run and maintain a specified number of instances of a task definition simultaneously in an Amazon ECS cluster. This is called a service. This controls how many containers to run, scaling policies, networking configuration, etc.

1. Click **Clusters**, then **click test-ecscluster-418023852230-us-east-2**.
2. Under the *Services* tab, click **Create**.
    - For *Launch Type*, click **FARGATE**.
    - For *Task Definition*, select your name for the "Family", then *1 (latest)* for the "revision".
    - For *Service name*, put in your first initial, last name.
    - For *Number of Tasks*, put in 1. Scroll all the way down, click **Next Step**.
    - Uncheck *Enable service discovery integration*, then scroll up a bit to the *VPC and security groups* section.
    - For *Cluster VPC*, select **vpc-08850982ee9a0f0a1**. For *Subnets*, select one with the word **PUBLIC** in the name, doesnt matter which. Ensure *Auto-assign public IP* is enabled.
    - Scroll all the way down and click **Next Step** twice, then **Create Service**.
    - You will be taken to the status screen for your service. Hit refresh on the right hand side until you see a task appear and go to the *last status* of **RUNNING**. This means your container is running

successfully. Click the **Task** link (long guuid-like number). Under the *Network* section, you will see a **Public IP**. Copy and paste it into your browser and you should see your container's web page!

Lambda LAB

This will create a S3 bucket which, when a file is dropped there, will trigger a lambda that will read what type of file it is and output the metadata to Cloudwatch.

Steps

1.) Go to S3 console and click “Create Bucket”. Name it 99999-YOURNAME, substituting in your first initial, last name. Click Create.

2.) Go to Lambda console and create a new function. Create from a blueprint and search for “s3”. Select the “s3-get-object-python” blueprint. Click “Configure”.

3.) Fill out as such:

- Name your function 99999-YOURNAME, like the s3 bucket above.
- Set EXECUTION ROLE to “Create a new role with basic Lambda Permissions”.
- Scroll down a bit and select your bucket create in step 1.
- Hit "Create Function"

4.) Go to your bucket and look at the properties tab. Scroll down to EVENTS....notice how it's now associated with the lambda function you create in step 2; that was done via the "Enable Trigger" checkbox. Go ahead and upload a file, doesn't matter what.

5.) Go to the Cloudwatch console, expand the "Logs" section, and click "Log groups" on the left hand side. In the filter box, look for "/aws/lambda/99999-YOURNAME".

6.) Once you find it, click it then click "Search Log Group". Did the lambda trigger successfully output the type of file we uploaded? Why or why not?

API Gateway LAB

This will create an API Gateway that sits in front of an SQS queue, allowing us to hit an authenticated endpoint and send a JSON body that will end up in a queue.

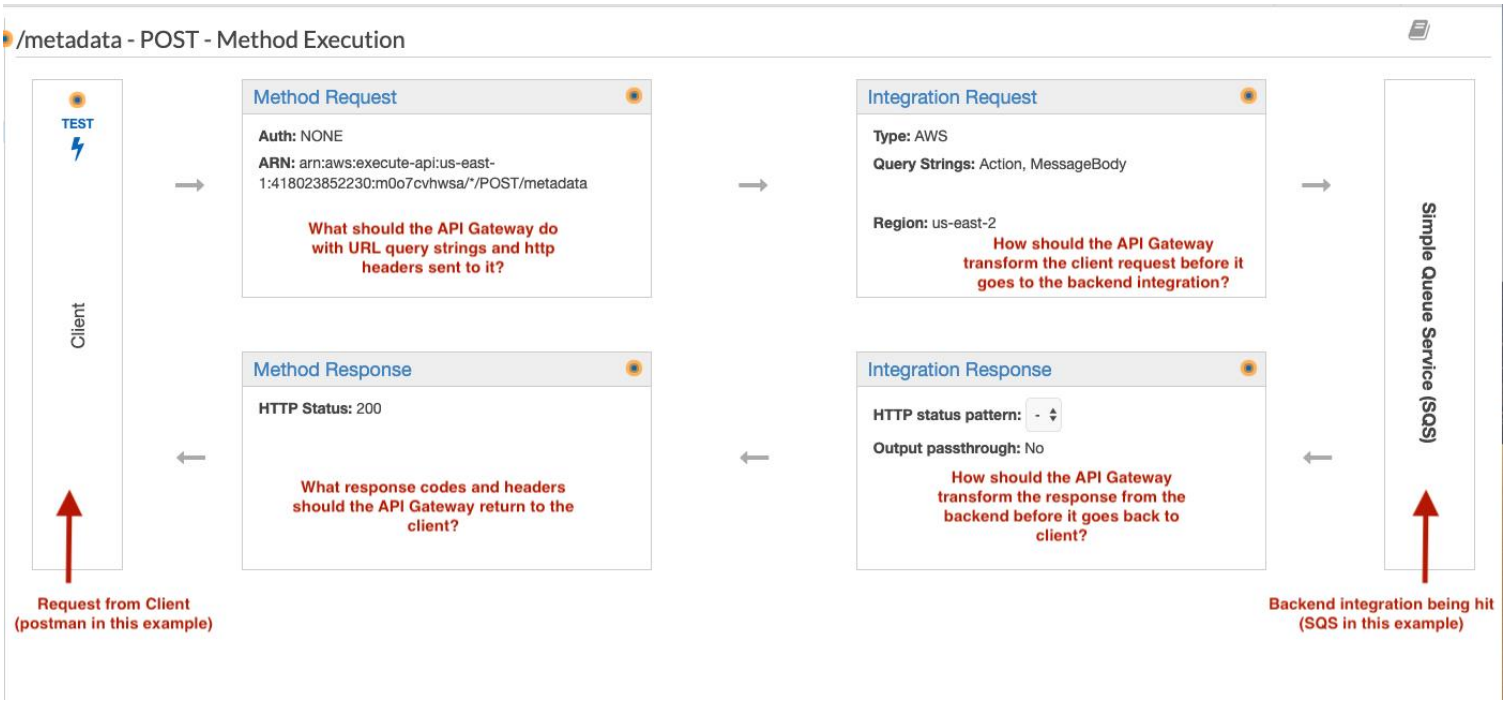
1.) Go to the SQS console and create a new queue. Call it your 99999-YOURNAME, where YOURNAME is your first initial,last name. Click **Quick-create queue**

2.) Navigate to https://git.rockfin.com/training-iac/training-starter-kit/blob/master/docs/AWS/swagger.json. Copy the contents of that file to your clipboard.

2.) GO to the API Gateway console in AWS. Click "Create API". Navigate to the "REST API" box and click **Import**.

3.) Select *Import from Swagger or OpenAPI 3*, then paste in the swagger you copied in step 1. Replace *YOURNAME* with your first initial, last name in the 2 spots where the placeholder is (line 5, 7 and line 21). Click **Import**.

4.) Click on the *POST* method under the */sendtosqs* resource. Now how you're presented with 6 boxes, 4 in the middle surround by long ones on the left and right. This shows the basic process for how an API Gateway functions. See screenshot below:

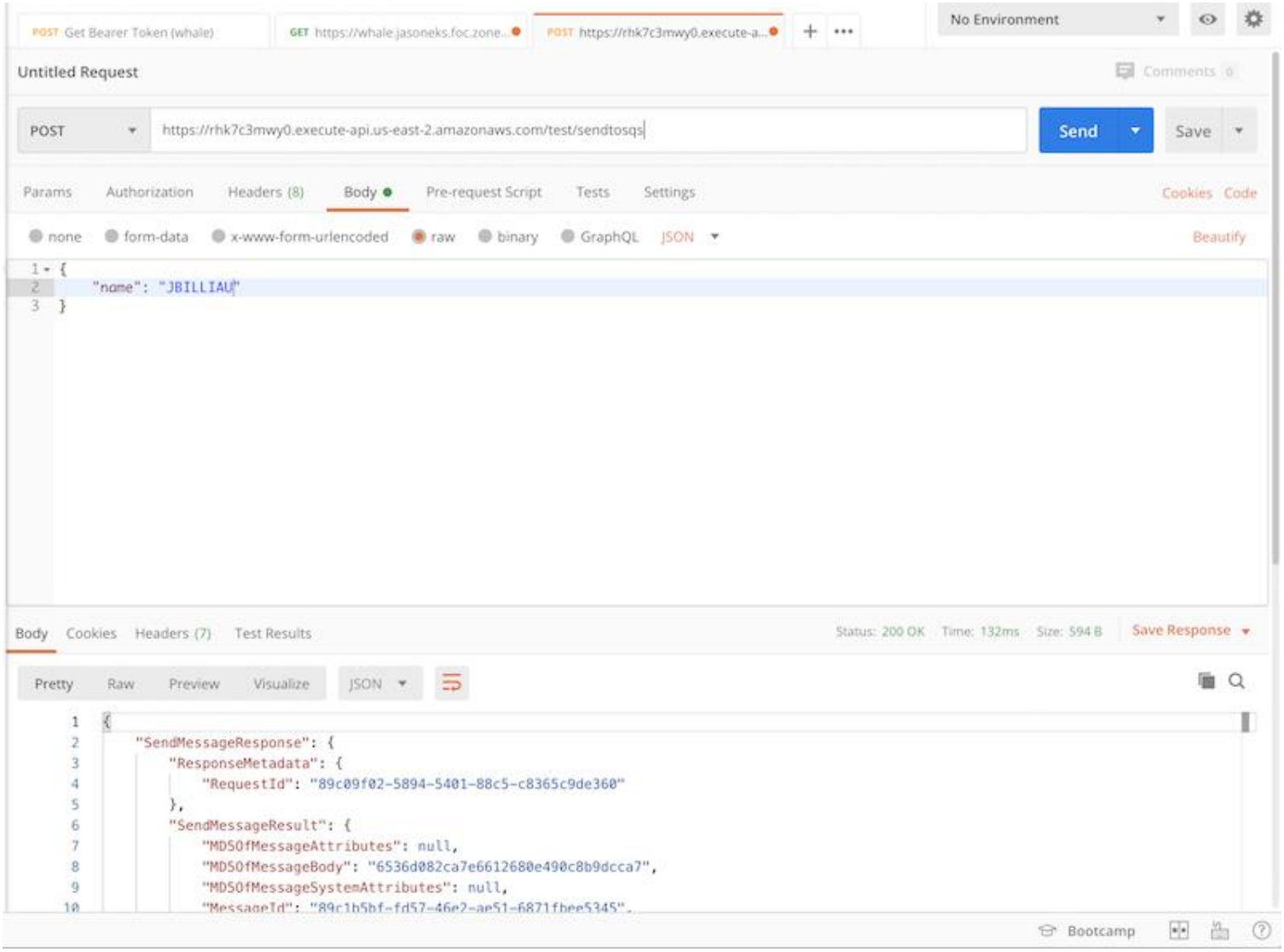


5.) Now that you understand the basic flow, let's deploy this and get it working! On the left-top side of the screen, click *Actions* and then **Deploy API**. For deployment stage, select "[New Stage]" and for Stage Name, put in *test*. Click **Deploy**.

6.) Note you now have an invoke url....your API Gateway is officially deployed and able to be used. Then do the following steps:

- Go into Postman and paste that URL into a new tab, appending "/sendtosqs" to the URL.
- Change your method to POST.
- Click "body" underneath the URL and paste in the following json, replacing YOURNAME. { "name": "YOURNAME" }
- Change "text" to application/json.

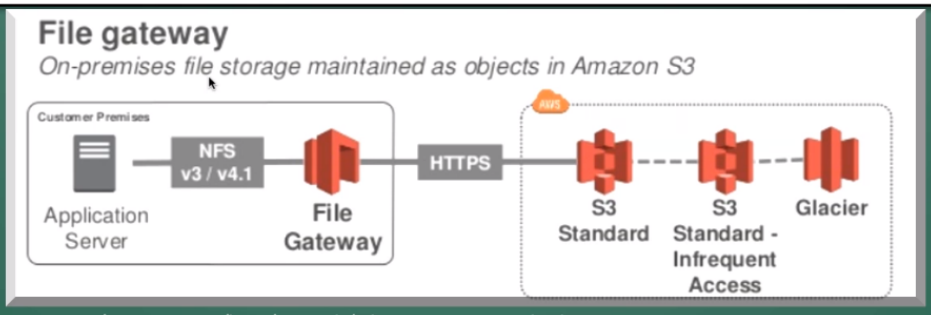
It should look like this:



7.) Hit **Send** in Postman and see if you get a 200 back. If so, go check your queue by selecting it, then click "Send and receive messages". Once the window opens, click "start polling for messages". Is your message there?

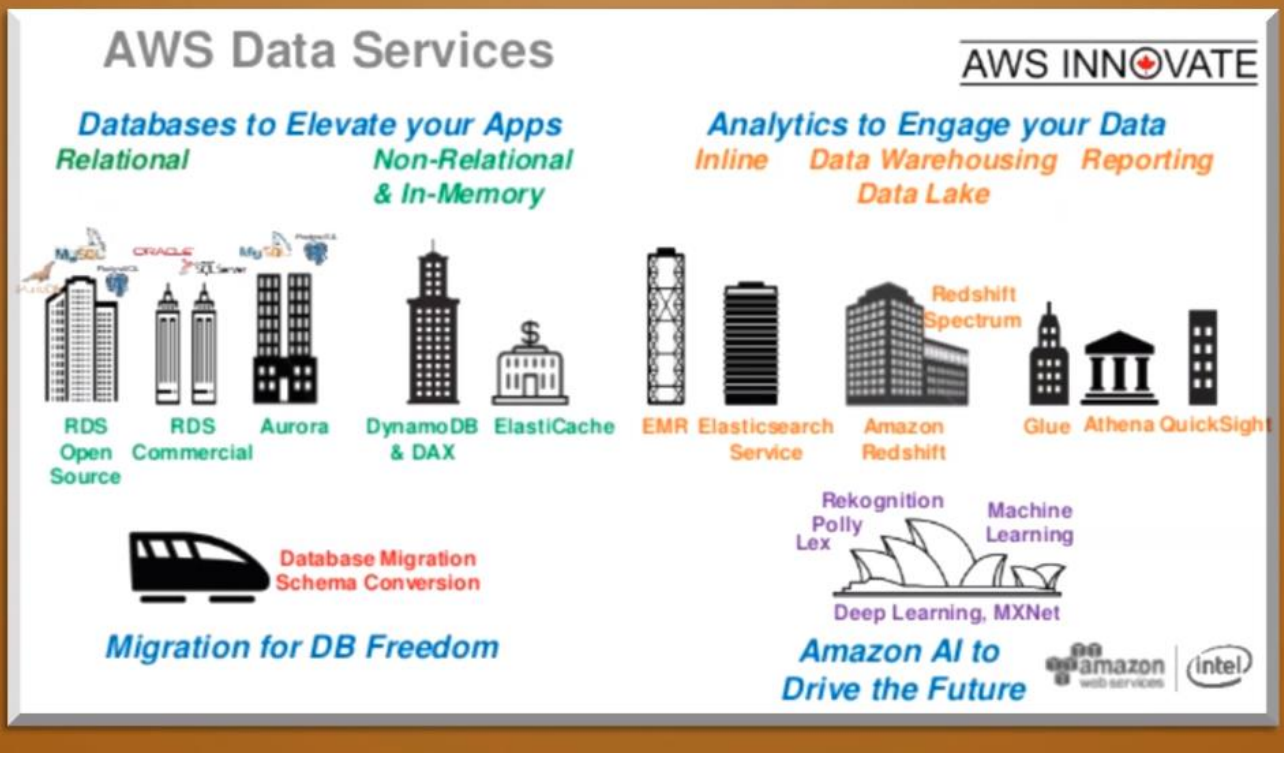
STORAGE GATEWAY

- Connect onprem applications to cloud storage (S3) via a VM you install in your datacenter
- Local caching for low-latency and quick access to frequently used data
- Using cloud file gateway, can present an S3 bucket as local file storage on an EC2.
- Consistent low latencies regardless of file system size
- Once the file gateway has moved data into Amazon S3, you can manipulate, analyze and manage it using native AWS services via API. Additionally, from your Amazon S3 bucket, you can distribute that data to other regions around the world with Cross-Region Replication, apply storage management tools, and use Lifecycle policies to migrate it to different storage S3 storage tiers



Article: https://docs.aws.amazon.com/storagegateway/latest/userguide/WhatIsStorageGateway.html

DATA SERVICES



RDS
RELATIONAL DATABASE SERVICE

What:

Type of service - Database Platform as a Service (PaaS)
Recommended QL Engines – MS SQL Server, Aurora (MySQL & PostgreSQL), and MySQL

Benefits:

Automatic backups
Multi-AZ high availability
Automated database patching
No infrastructure to manage and minutes for deployment
Enables encryption at rest and in transit
Simplifies scaling

Some Limitations:

Storage limits – 16TB for MS SQL Server, 64 TB for Aurora (2-4-2019)
No Access to host operating system.
No access for Super User/System Admin

Migration Assistance:

1. Schema Conversation Tool – Converts your schema from one engine to another engine
2. Data Migration Service – migrates data from onprem server to cloud RDS and/or engine to different engine.

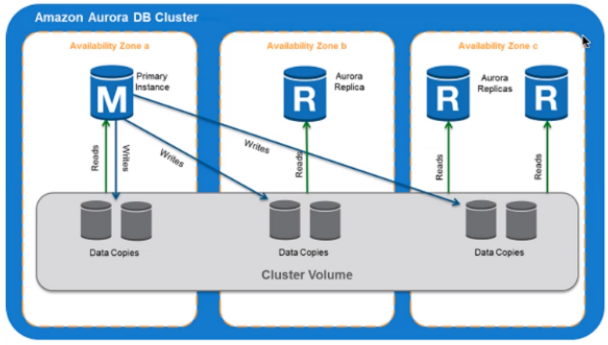
**No perfect tool to convert/migrate everything. Manual engineering may still be required.

Article: https://docs.aws.amazon.com/AmazonRDS/latest/UserGuide/Welcome.html

RDS
RELATIONAL DATABASE SERVICE

Aurora has many advantages over MS SQL Server on RDS:

- Exorbitantly less expensive
- You can create up to 15 replicas, which increases throughput and is more than enough failover targets. These replicas share storage with the primary instance which almost eliminates the lag in replication. All of Aurora's read replicas can be accessed independently; for SQL Server, we cannot access the stand by read replica. (unless MSSQL EE edition, VERY $$$$.
- 6 copies of your data, spread across 3 AZ's
- Supports multi-region via Global Databases
- Supports backtrack – ability to rollback to a point-in-time up to the second (basically an undo button for a DB)
- When initiating a backtrack, Aurora will pause the database, close any open connections, drop uncommitted writes, and wait for the backtrack to complete. Then it will resume normal operation and be able to accept requests.



HOWEVER, It's not a complete slam dunk:

1.Performance: MS SQL has smarter query analyzer and in order to take good Aurora (MySQL) performance you need to understand how it works very well. SQL Server can "excuse" a lot of developer things.

2.Development: SQL Server has more professional development tools like SQL Server Management Tool, SQL Profiler, and Tuning Adviser and so on.

Article: https://docs.aws.amazon.com/AmazonRDS/latest/UserGuide/Welcome.html

DYNAMODB

- Amazon DynamoDB is a NoSQL, schema-less, key-value and document database that can deliver single-digit millisecond performance at any scale. Support multi-region capability with global tables.
- Instead of retrieving data with complex SQL queries, data is read/written with HTTP REST calls. Effectively turning a database into something that operates like any other web api.
- To create a table, we just define the primary key. Items can be added into these tables with a dynamic set of attributes. *Items* in DynamoDB correspond to *rows* in SQL, and *attributes* in DynamoDB correspond to *columns* in SQL. DynamoDB supports the following data types:
  - **Scalar data types**: *Number*, *String*, *Binary*, *Boolean* - **Collection data types**: *Set*, *List*, *Map*
- DynamoDB supports GET/PUT operations by using a user-defined primary key. The primary key is the only required attribute for items in a table.
- Throughput measured in read capacity units (RCU) and write capacity units (WCU).
  - A *read capacity unit* represents one strongly consistent read per second, or two eventually consistent reads per second, for an item up to 4 KB in size.
  - A *write capacity unit* represents one write per second, for an item up to 1 KB in size.

Article: https://docs.aws.amazon.com/amazondynamodb/latest/developerguide/Introduction.html

DYNAMODB (CONTINUED)

DynamoDB has two types of keys and when you select a key type and you can't change once it is selected:

- Simple key – in this case, you need to identify what attribute in the table contains a key. This key is called a partition key. With this key type, DynamoDB does not give you a lot of flexibility and the only operation that you can do efficiently is to store an element with a key and get an element by a key back.
- Composite key – in this case, you need to specify two key values which are called partition key and a sort key. As in the previous case, you can get an item by key, but you can also query this data in a more elaborate way. For example, you can get all items with the same partition key, sort result data by the value of the sort key, filter items using the value of the sort key, etc. The pair of partition/sort should be unique for each item.

With this table, the only operation that we can perform efficiently is to get a user by id.

ID	Email	Name
1	joe@example.com	Joe
2	peter@example.com	Peter
3	anonmous@example.com	Kyle

Partition key

Composite keys allow more flexibility. This structure would allow performing more complex queries like:

- Get all forum posts written by a specified user sorted by time (we can do this because we have the sort key)
- Get all forum posts that were written in a specified time interval (we can do this because we can specify filtering expression on a sort key)

Partition key

ID	Timestamp	Message
1	100	Lunch time?
1	200	Just had my sandwich
1	300	This forum is boring...
2	150	Hello everybody
3	250	Kitten photos here:

Sort key

Article: https://docs.aws.amazon.com/amazondynamodb/latest/developerguide/Introduction.html

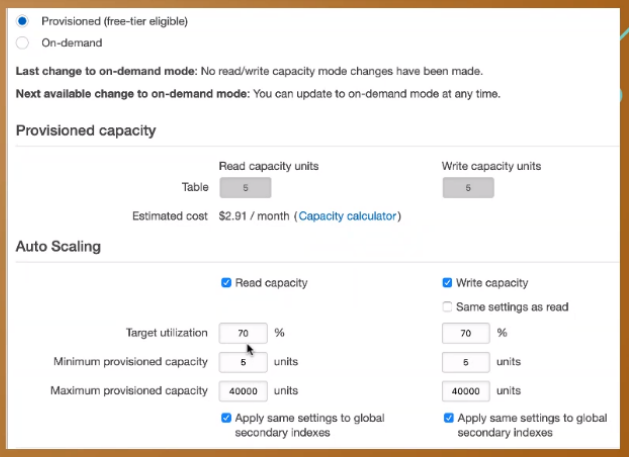
DYNAMODB - AUTOSCALING

All you need to do is to enable autoscaling on a particular table, and AWS will automatically increase or decrease provisioned capacity depending on current load. There are 2 modes: **Provisioned Capacity and Ondemand Capacity.**

- **Provisioned Capacity** - how many and how large of reads and writes you could make on your table in any given second. Read and write capacity units are charged by the hour, and your requests will be throttled if you exceed your provisioned capacity in any given second. To set correctly, need to know your average item size and expected read/write request rates.
- **Ondemand Capacity** – Yeah just make it scale, thanks.

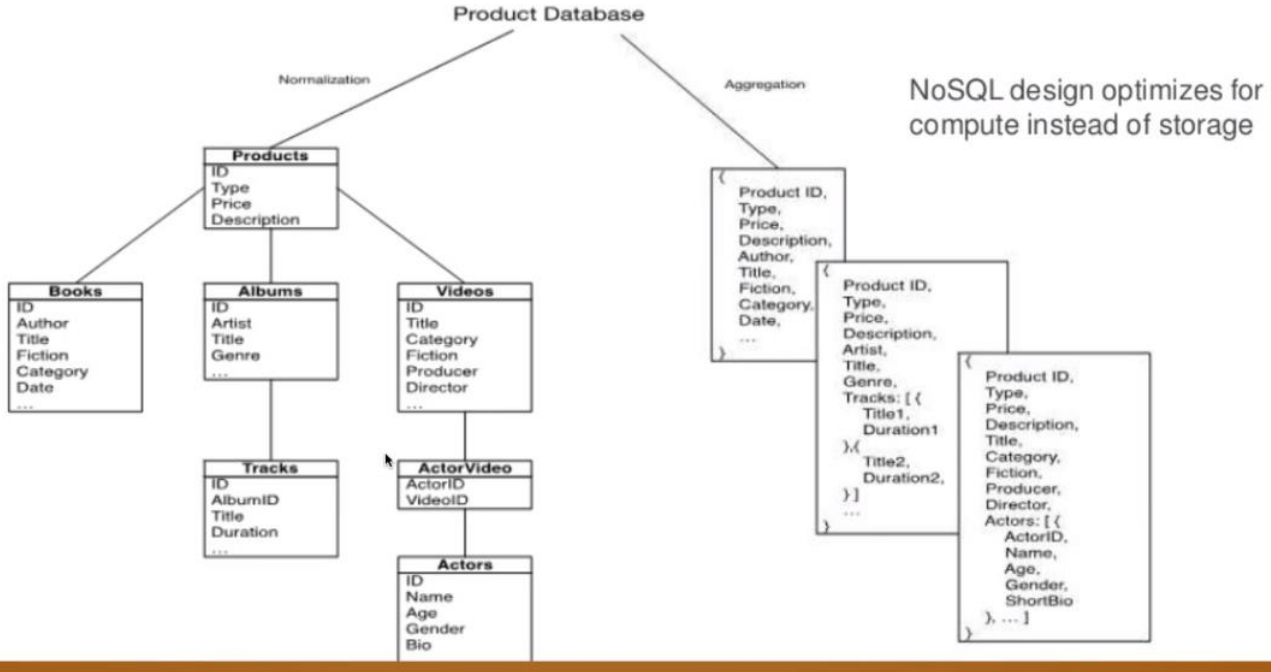
If you have *steady, predictable traffic*, **choose provisioned capacity**. Since you know you need a certain amount of capacity at all times, you can save from reduced rates.

If you have *variable, unpredictable traffic*, **choose on-demand.** If your application gets random spikes, it can be hard to provision capacity to match demand. Use the on-demand feature so you don't throttle your users.



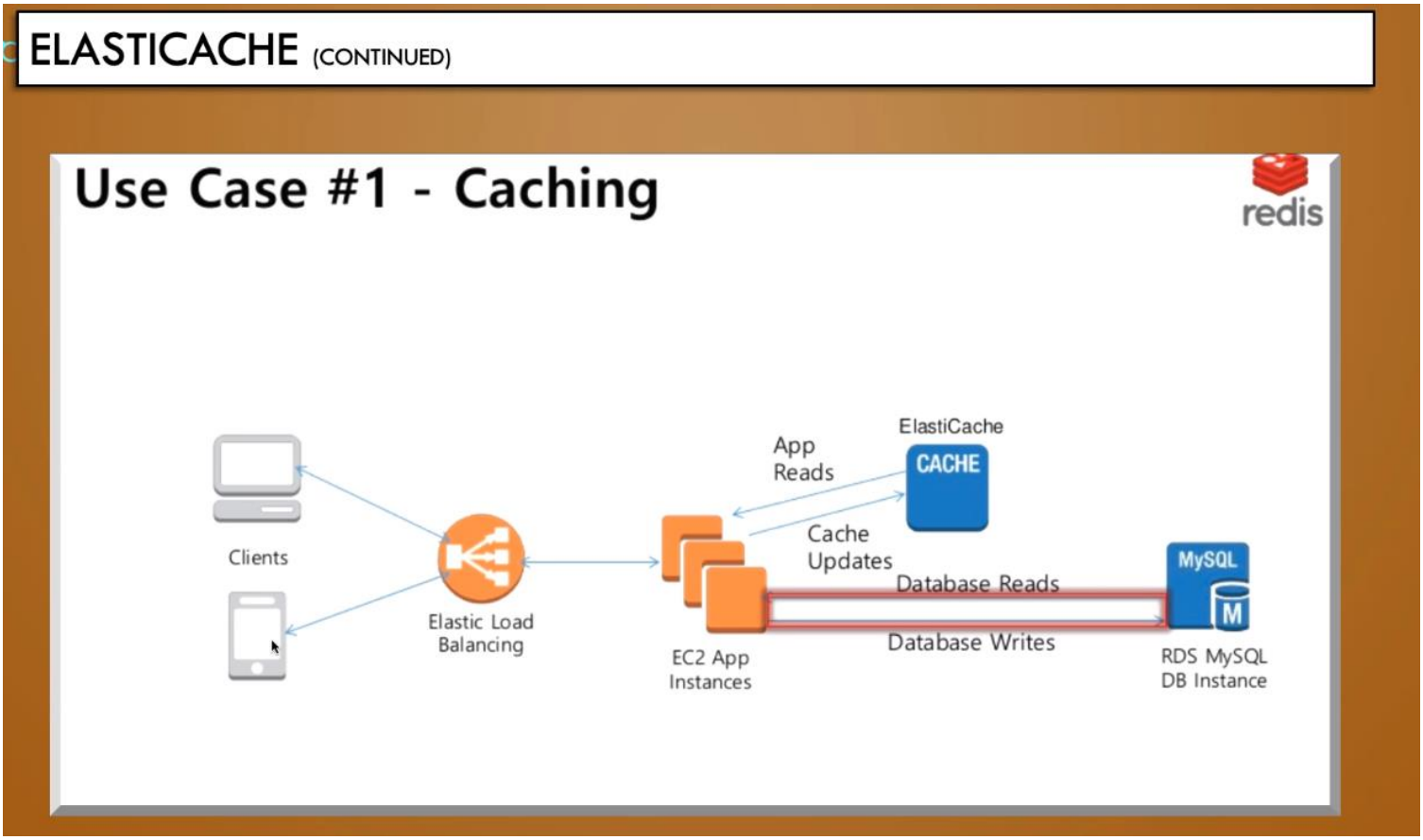
DYNAMODB (CONTINUED)

SQL vs. NoSQL schema design

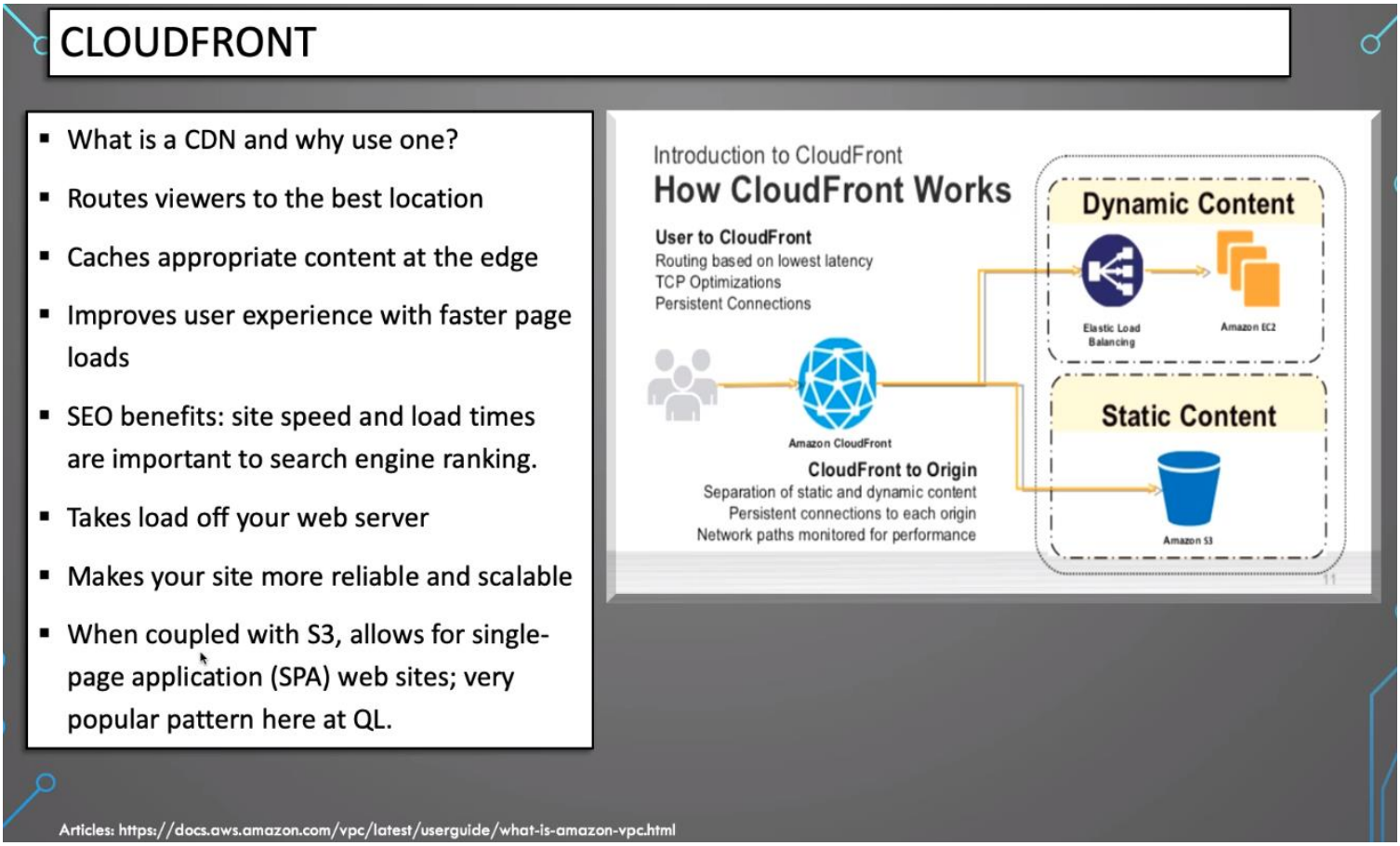


Article: https://docs.aws.amazon.com/amazondynamodb/latest/developerguide/Introduction.html

If we are using mysql for service providers, we could use Easticcache when not many updates are created.

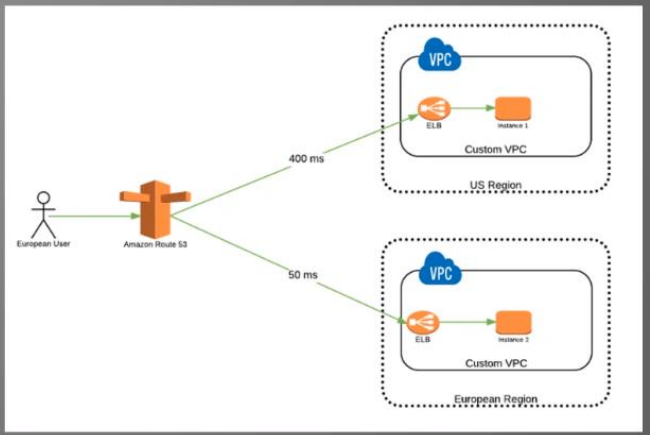


New speed increase for SP - Formula: to get more speed of SP. application -> elastic cache -> mysql/aurora db.. clear cache with lambda trigger, whenever data is modified. Because SP updates are not frequent.



ROUTE 53

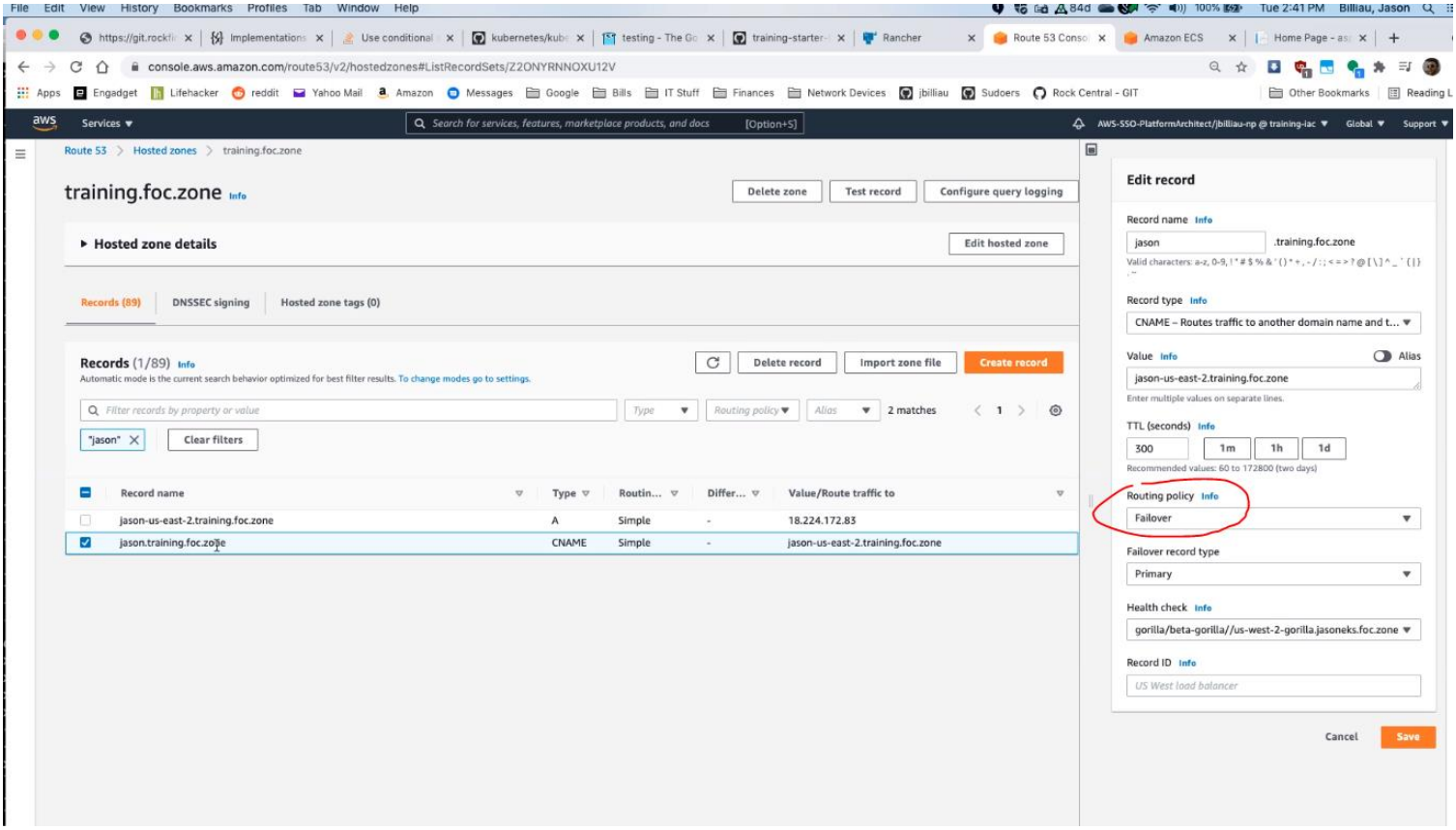
- Hosted DNS in AWS
- Register domain names
- Route internet traffic to resources for your domain - Provides friendly names for API Gateway, ELB, etc.
- Understanding record sets which are available A, CNAME, etc
- Support multiple routing policies: simple, weighted round robin, latency-based, and geolocation.



- Able to do health checks on endpoints. Once enabled, health check agents will monitor each end-point of your application to determine its availability. If unhealthy, failover can be configured to do alternate endpoint. Makes multi-region possible.

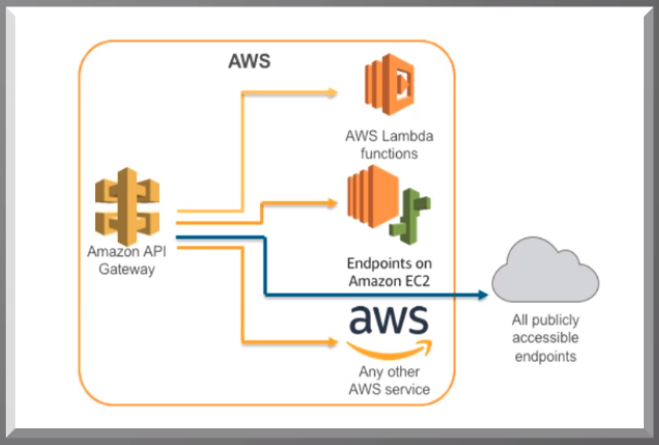
Article: https://docs.aws.amazon.com/Route53/latest/DeveloperGuide/Welcome.html

Do we have failover policy for reggieviewer:



API GATEWAY

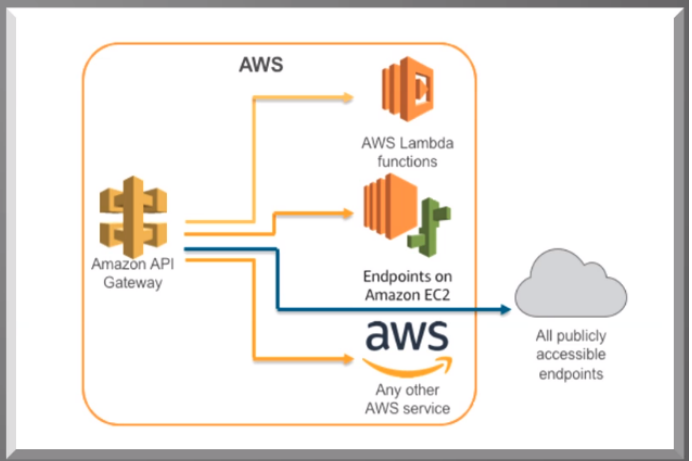
- Are HTTP-based.
- Enable stateless client-server communication.
- Implement standard HTTP methods such as GET, POST, PUT, PATCH, and DELETE
- Host multiple version and stages of your API
- Create and distribute API keys to enable throttling and quotas
- Swagger support
- Can be used to authenticate requests into backend services. For example, we use API Gateways with a Lambda proxy, authorized with a Lambda authorizer as well, to get requests to backend ECS containers.
- GET to POST
  - Read all query string parameters from your GET request, create a body to make POST to your backend
- XML to JSON
  - Receive XML input, transform to JSON for backend



Article: https://docs.aws.amazon.com/apigateway/latest/developerguide/welcome.html

API GATEWAY

- Are HTTP-based.
- Enable stateless client-server communication.
- Implement standard HTTP methods such as GET, POST, PUT, PATCH, and DELETE
- Host multiple version and stages of your API
- Create and distribute API keys to enable throttling and quotas
- Swagger support
- Can be used to authenticate requests into backend services. For example, we use API Gateways with a Lambda proxy, authorized with a Lambda authorizer as well, to get requests to backend ECS containers.
- GET to POST
  - Read all query string parameters from your GET request, create a body to make POST to your backend
- XML to JSON
  - Receive XML input, transform to JSON for backend

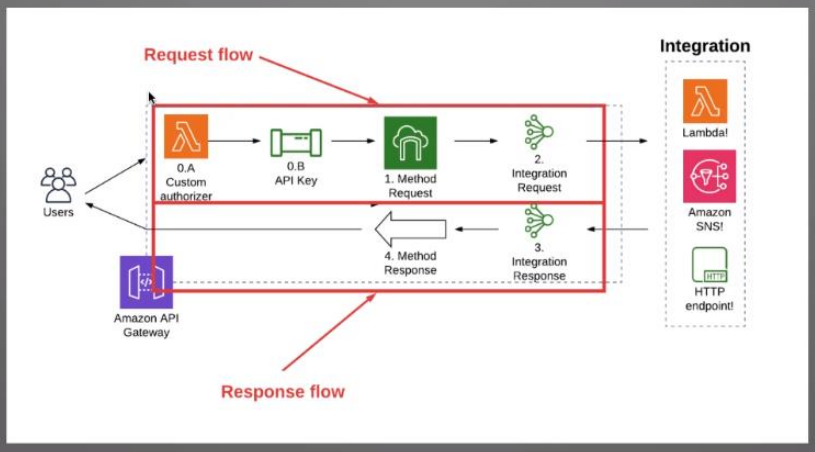


Article: https://docs.aws.amazon.com/apigateway/latest/developerguide/welcome.html

API GATEWAY

When an HTTP request comes to API Gateway, it will go through three elements:

- First, it will go through the request flow to authorize, validate, and transform the request.
- Second, it will go to the integration where the request will be handled by your service.
- Finally, it will go through the response flow to transform and prepare the response for the original client.

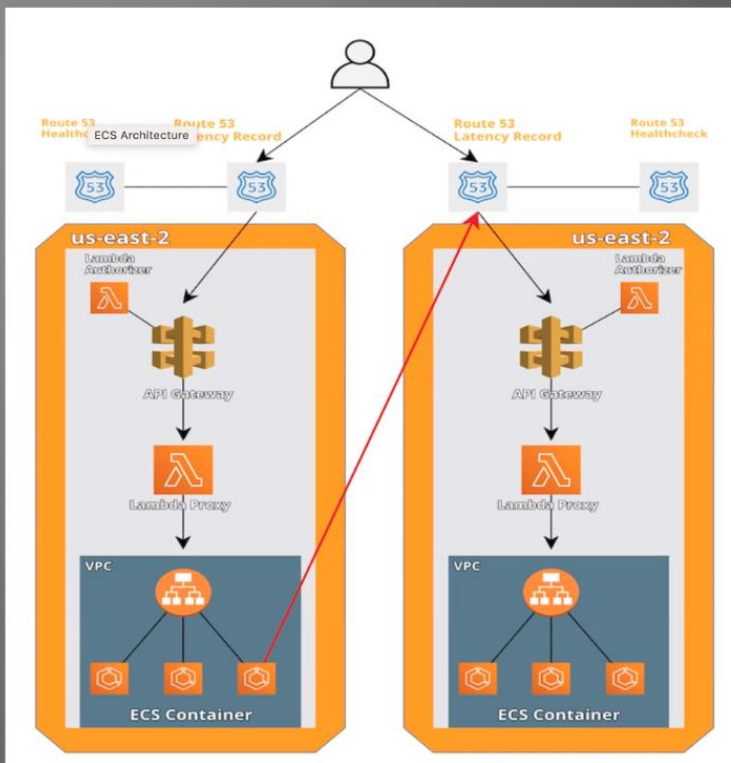


Article: https://docs.aws.amazon.com/apigateway/latest/developerguide/welcome.html

API GATEWAY

VERY common architecture here at QL.
Let's walk through it!

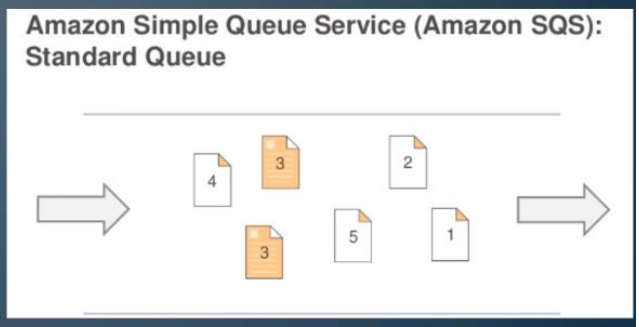
1. Traffic lands on API Gateway, which has a custom domain name aliased to a friendly name via Route53
2. Lambda authorizer launches, reads *Authorization* header which contains bearer token, calls PingFed and validates token and requested scopes
3. If request passes authorizer, API Gateway passed it to Lambda proxy, which tunnels into VPC and hits backend application load balancer.
4. Attached to that load balancer is a target group.
5. In that target group are ECS containers, which get round robined across, scaling up and down automatically to meet a target tracking CPU average (default 70%).

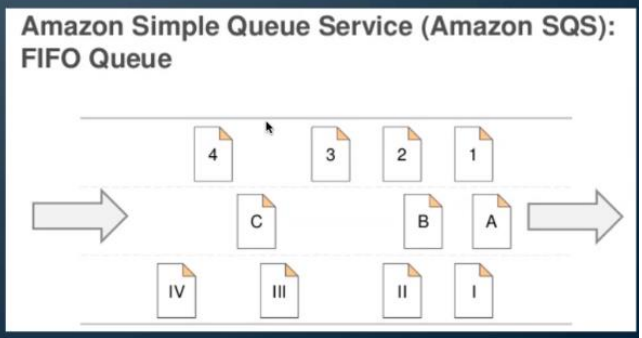


Article: https://docs.aws.amazon.com/apigateway/latest/developerguide/welcome.html

SQS
SIMPLE QUEUE SERVICE

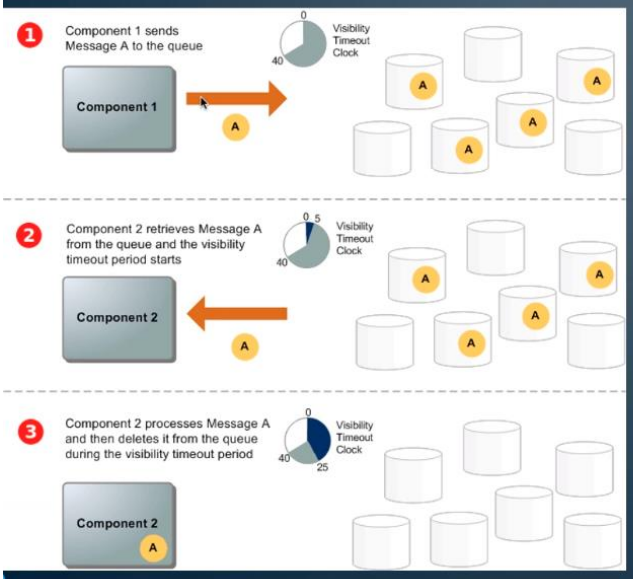
- Message queuing service (UM and Sonic equivalent)
- Types of Queues available:
  Standard Queues - FIFO (First-In-First-Out)
  Dead Letter Queues - Delay Queues
- Maximum payload of 256KB
- Unlimited backlog up to 14 days
- Unless you're using a FIFO queue, SQS provides "at-least-once" delivery, which means you might get the same message more than once.
- Making your compute layer idempotent (i.e. it can run more than once for the same queue message with no unacceptable side effects) is recommended to solve any issues arising from duplicate messages.

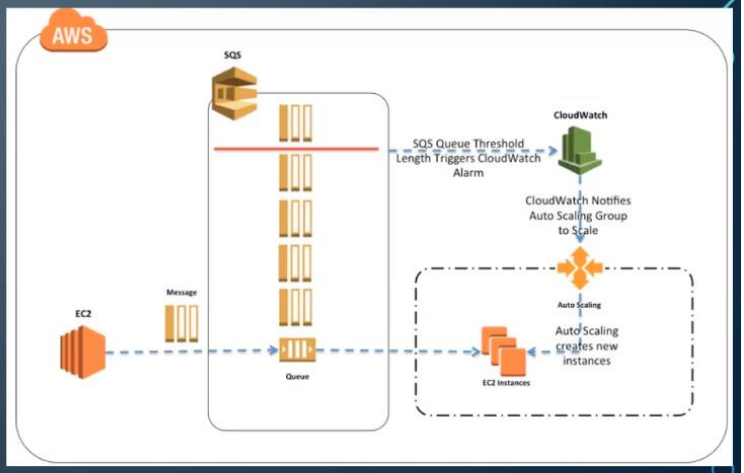




Articles: https://docs.aws.amazon.com/AWSSimpleQueueService/latest/SQSDeveloperGuide/welcome.html

SQS (CONTINUED)
SIMPLE QUEUE SERVICE





Articles:
https://docs.aws.amazon.com/AWSSimpleQueueService/latest/SQSDeveloperGuide/welcome.html

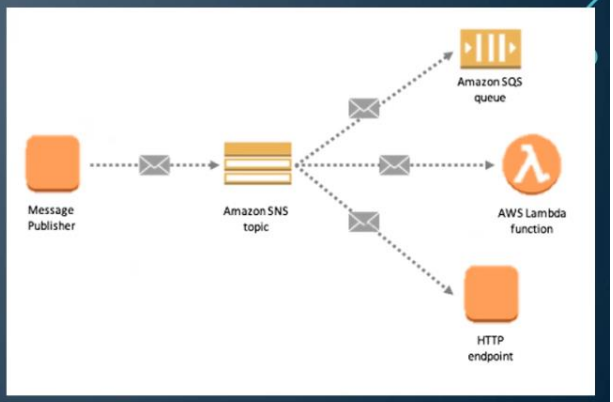
Simple Notification Service:

SNS
SIMPLE NOTIFICATION SERVICE

- Subscription based service
- Event-driven computing hub:

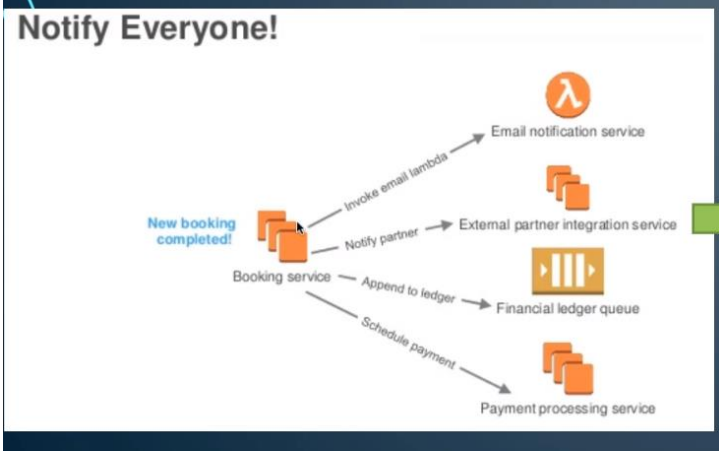
Event Sources	Event Destinations
EC2	SQS
S3	Lambda
RDS	HTTP endpoints
Lambda	Email

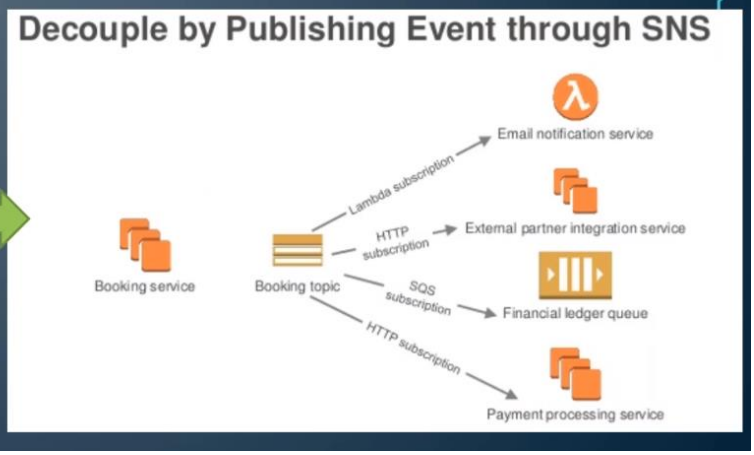
- SNS makes it simple and cost-effective to send out push notifications to mobile devices, email recipients, or send messages to other AWS services and endpoints.
- Popular approach is called fan in > fan out, where multiple applications can hit one SNS topic (fan in) that then goes and triggers multiple event destinations (fan out).



Article:
https://aws.amazon.com/sns/

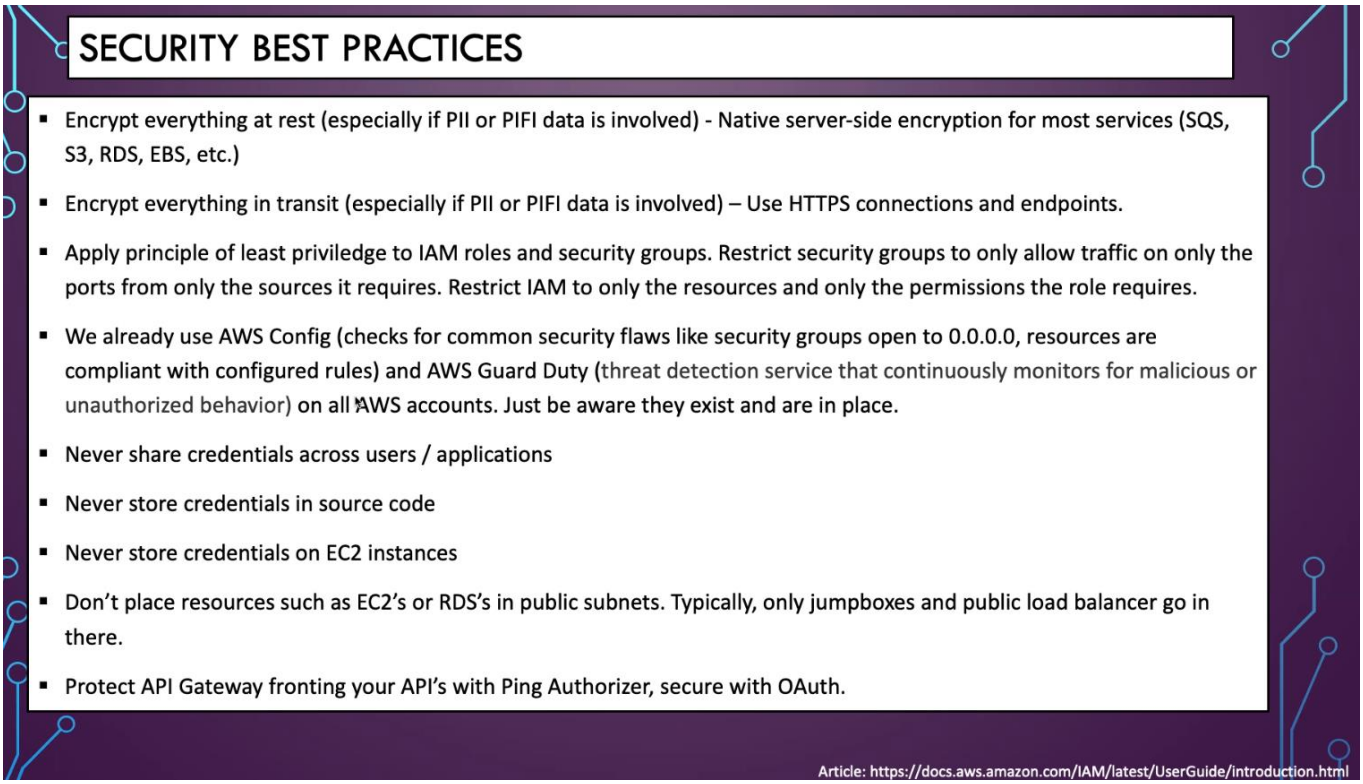
SNS (CONTINUED)
SIMPLE NOTIFICATION SERVICE





Security:


SECURITY
Ask yourself some questions:
1. How will your service will be accessed (public or private)?
2. What sort of data are you handling?
3. Are there any regulations you need to be compliant with?
4. Are there any compliance assessments you need to plan for?
5. Who will be administering the application?
6. Who needs to audit the platform (internal or external)?
aws


SECURITY BEST PRACTICES
Encrypt everything at rest (especially if PII or PIFI data is involved) - Native server-side encryption for most services (SQS, S3, RDS, EBS, etc.)
Encrypt everything in transit (especially if PII or PIFI data is involved) – Use HTTPS connections and endpoints.
Apply principle of least priviledge to IAM roles and security groups. Restrict security groups to only allow traffic on only the ports from only the sources it requires. Restrict IAM to only the resources and only the permissions the role requires.
We already use AWS Config (checks for common security flaws like security groups open to 0.0.0.0, resources are compliant with configured rules) and AWS Guard Duty (threat detection service that continuously monitors for malicious or unauthorized behavior) on all AWS accounts. Just be aware they exist and are in place.
Never share credentials across users / applications
Never store credentials in source code
Never store credentials on EC2 instances
Don't place resources such as EC2's or RDS's in public subnets. Typically, only jumpboxes and public load balancer go in there.
Protect API Gateway fronting your API's with Ping Authorizer, secure with OAuth.
Article: https://docs.aws.amazon.com/IAM/latest/UserGuide/introduction.html

Logging and Watching Logs:

CLOUDWATCH

- CloudWatch provides you with data and actionable insights to monitor your applications, understand and respond to system-wide performance changes, optimize resource utilization, and get a unified view of operational health
- Collects
  - Easily collect and store logs
  - Built in metrics for almost all AWS services
- Monitors
  - Can setup dashboards to view metric data
  - Alarms allow you to set a threshold on metrics and trigger an action
- Act
  - Set a threshold to alarm on a key metric and trigger an automated Auto Scaling action
  - Write rules to indicate which events are of interest to your application and what automated actions to take when a rule matches an event
- Can store logs in S3 for long-term storage or send to Elasticsearch for search/analyzing.
- Cloudwatch Events can trigger Lambda's on a set schedule, replicating cron type functionality.

Article: https://docs.aws.amazon.com/AmazonCloudWatch/latest/monitoring/WhatIsCloudWatch.html

COSTS

- Everything you spin up in AWS costs….money!
- Provision only what you need, scaling higher or wider as demand requires.
- Scale intelligently based off what load you expect and have historical trends backing that up. No, you don't need to handle 1,000,000 requests an hour when or the past 3 years you've averaged 600.
- AWS Billing Dashboard is a great way to track down down to the penny what your account is costing and what services encompass those costs. cloudhub.foc.zone is also an internal website we have that is more open to virtually everyone and should have the same, if not similar, data.
- Everything is IAC now; if the project you were working on is backburnered, tear down your infrastructure. Done testing? Tear down your infrastructure? Going on maternity/paternity leave for 3 months and have infrastructure stood up for a POC? TEAR IT DOWN.
- Cloudwatch and Grafana are both great options for seeing if the infrastructure you provisioned is suitable for your particular use-case. Provision a c5.2xlarge and CPU has been averaging 6% over past 3 months? You can probably go down to a much lower instance type.
- Overall, just be cognizant of what you are standing up, how much it costs, and make sure you are making the right decision for your application; don't just throw money at problems.

Deploying Infrastructure with Terraform and HAL

Core Concepts

Why Infrastructure as Code?

How do we deploy Terraform at FOC?

- HAL and what that means. Terragrunt, state management, builds and deploys

Standard Modules at FOC.

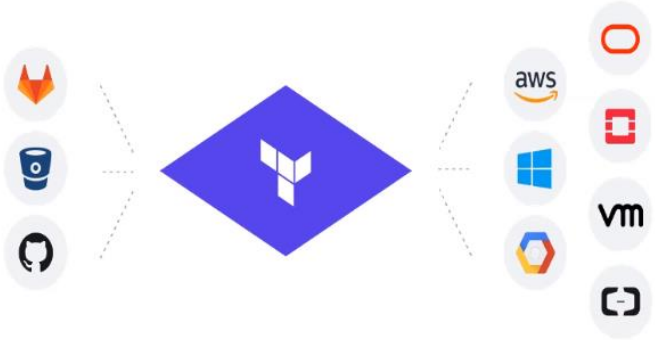
- When and how to use the standard modules. What other options are available?
- The concept of keeping our modules DRY

Variable Declaration, Population and Overriding.

- Concepts most important for consumers who interact with Terraform at FOC.
- What role the **terraform.tfvars** file plays and how it is used to deploy Terraform modules in our environment.

What Is Terraform?

- Open source tool to provision infrastructure for private cloud, public cloud and external services.
- Enables the creation of reusable modules to define the infrastructure topology with code.
- Uses Hashicorp developed declarative language HCL.
- Written in Go. Allows for additional providers to be easily integrated.



Why Terraform?

- Standardization, version control, and visibility
- Speed (deployment, changes, patches, upgrades)
- Deployment delegation

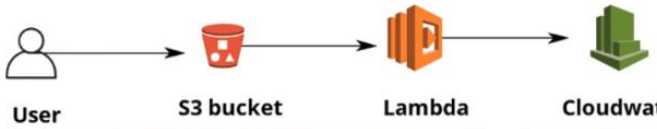
Why Terraform over CloudFormation?

- Reusable.
- Multiple providers allows for use in any cloud platform and many non-cloud related tools.

Modularity

- Reduces the knowledge required to deploy consistent, secure and efficient Infrastructure.
- "Lego Block" approach simplifies complex infrastructure and application architectures

Terraform Example




```
resource "aws_iam_policy" "lambda_logging" {
  name = "lambda_logging"
  path = "/"
  description = "IAM policy for logging from a lambda"

  policy = <<EOF
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Action": [
        "logs:CreateLogGroup",
        "logs:CreateLogStream",
        "logs:PutLogEvents"
      ],
      "Resource": "arn:aws:logs:*:*:*",
      "Effect": "Allow"
    }
  ]
}
EOF
}
```

```
resource "aws_iam_role_policy_attachment" "lambda_logs" {
  role       = "${aws_iam_role.iam_for_lambda.name}"
  policy_arn = "${aws_iam_policy.lambda_logging.arn}"
}

1 references
resource "aws_s3_bucket" "default" {
  bucket = "file-extensions-bucket"
}

0 references
resource "aws_s3_bucket_notification" "bucket_notification" {
  bucket = "${aws_s3_bucket.default.id}"

  lambda_function {
    lambda_function_arn = "${aws_lambda_function.test_lambda.arn}"
    events              = ["s3:ObjectCreated:*"]
  }
}

1 references
resource "aws_iam_role" "iam_for_lambda" {
  name = "iam_for_lambda"

  assume_role_policy = <<EOF
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Action": "sts:AssumeRole",
      "Principal": {
        "Service": "lambda.amazonaws.com"
      },
      "Effect": "Allow"
    }
  ]
}
EOF
}
```

```
resource "aws_lambda_function" "test_lambda" {
  function_name = "${var.lambda_function_name}"
  filename      = "sample.zip"
  role          = "${aws_iam_role.iam_for_lambda.arn}"
  handler       = "exports.example"
  runtime       = "go1.x"
  depends_on    = [
    "aws_iam_role_policy_attachment.lambda_logs",
    "aws_cloudwatch_log_group.example"
    ]
}

resource "aws_lambda_permission" "allow_bucket" {
  statement_id  = "AllowExecutionFromS3Bucket"
  action        = "lambda:InvokeFunction"
  function_name = "${aws_lambda_function.test_lambda.arn}"
  principal     = "s3.amazonaws.com"
  source_arn    = "${aws_s3_bucket.bucket.arn}"
}


resource "aws_cloudwatch_log_group" "example" {
  name              = "/aws/lambda/${var.lambda_function_name}"
  retention_in_days = 14
}
```

Deploying The Infrastructure

Terraform Init

- Initializes the directory for use with terraform.
- Pulls both local and remote configurations, initializes the backend, and installs plugins.
- First command run and safe to re-run. Will never destroy any data locally or at provider.

Terraform Plan (HAL Build)

- Preview of the infrastructure the code is about deploy. Time to review the changes that will be made.
- Also a chance to check for compatibility of changes to modules you may be using.

Terraform Apply (HAL Push)

- Deployment of the infrastructure code to AWS or whatever provider in use.
- Terraform does not roll back. It will fail on error. It is expected that the code is corrected and pushed again.

Modules

- Modules in Terraform are folders with Terraform files. That is it.
- The current working directory holding the Terraform files you're applying comprise what is called the *root module*. This itself is a valid module.
- Modules can be robust or very granular. At the FOC we try to stay as granular as it makes sense.
- Modules should follow a standard structure (at minimum):

```
$ tree minimal-module/
.
├── README.md
├── main.tf
├── variables.tf
├── outputs.tf
```

Modules are the foundation for reusable infrastructure as code.

Variables

- Input variables are the parameters for Terraform modules.
- We can pass variables from the root module to the child modules.
- Variable type constraints are defined by type keywords and type constructors.
  - Supported type keywords include string, number and bool
  - Supported type constructors are list, set, map, object and tuple

```
variable "image_id" {
  type = string
}
```

```
variable "availability_zone_names" {
  type    = list(string)
  default = ["us-west-1a"]
}
```

```
variable "docker_ports" {
  type = list(object({
    internal = number
    external = number
    protocol = string
  }))
  default = [
    {
      internal = 8300
      external = 8300
      protocol = "tcp"
    }
  ]
}
```

Variable Definition Files

.tfvars files are used to populate input variables. terraform.tfvars and *.auto.tfvars files in working directory will automatically populate variables.

Lab 1
https://git.rockfin.com/training-iac/training-starter-kit/blob/master/docs/terraform/LAB1.md

Configuring Your Module

Navigate to your training kit folder workshop_materials/terraform/example-module

Open the main.tf file and populate the file with the contents below. (If your file is not empty, wipe it and paste in the config below)

https://git.rockfin.com/training-iac/training-starter-kit/blob/master/workshop_materials/terraform/example-module/main.tf

```
provider "aws" {
  region     = var.aws_region
}

resource "aws_instance" "example" {
  ami                         = "ami-0cd3dfa4e37921605"
  instance_type               = var.instance_type
  associate_public_ip_address = false
  key_name                    = "tf-training"
  subnet_id                   = "subnet-046134d61fe120cbc"

  tags = {
    Name = "${var.application_name}-ec2"
  }
}
```

Next we'll establish the variables being used in the main.tf. The required variables to be populated in the variables.tf are below:

Open the variables.tf file and establish the variables.

The variable definition format is: **variable "variable_name" {}**
example:

```
variable "aws_region" {}
variable "application_name" {}
variable "instance_type" {}
```

Again, we are simply establishing the variables here, not setting values for them. We will be populating the variables with a terraform.tfvars file which allows you to pass variables to a terraform module or configuration, without modifying the code itself.

Now we need to define the variables in a variable input file: terraform.tfvars

Open the terraform.tfvars file and set the variable values.

This format is simple. **variable = "variable_value"** example:

```
aws_region       = "us-east-2"
application_name = "YOURNAME"
instance_type    = "t3.micro"
```

Make sure you replace YOURNAME

Deploying Your Infrastructure

Once you have the module configured, it is time to deploy the infrastructure you've defined.

We must first ensure we have the correct AWS credentials set.

These are the same we used in the AWS class to push our Docker image. If you still have this as your default credential, no changes are needed here.

There are a couple ways to configure credentials for Terraform. You can configure the AWS CLI default credentials, use environment variables, specify them in the provider block either explicitly or with a aws cli profile reference.

For our class we'll configure the default credentials:

AWS Configure

```
aws configure
AWS Access Key ID [None]: $FROM_LINK_ABOVE
AWS Secret Access Key [None]: $FROM_LINK_ABOVE
Default region name [None]: us-east-2
Default output format [None]: hit enter
```

Deploy Module

Once configured, in the example-module directory, run the following commands:

```
terraform init
```

Take note of the output. This initializes the working directory for terraform use. If initialize is clean, proceed.

```
terraform plan
```

Take note of the output. This details what resources and configurations will be deployed. If something doesn’t look right, correct it in the main.tf and run the plan again. Once the plan looks good, proceed.

```
terraform apply
```

Once again, take note of the output to ensure it matches the plan command and enter “yes” to deploy the infrastructure.
After you've verified the instance is up with your desired configuration, come back to the terraform.tfvars file and change the instance type from t3.micro to t3.small. Run through the plan and apply steps again to deploy the configuration change.

Destroying Your Infrastructure

The last step is to clean up our infrastructure once we've verified it deploys correctly.

To destroy the infrastructure, from the Terraform configuration root directory, run:

```
terraform destroy
```

Like the apply command, the destroy will output each resource it is going to destroy and prompt for confirmation before doing so. If everything looks as expected, enter YES and destroy the infrastructure.

Hal configuration and using Hal

What is HAL?

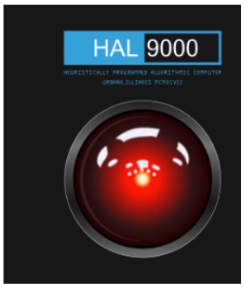
- QL Developed Code Deployment tool.
- Historically Linux Server Deploy Tool/Orchestrator
- Has transitioned into a Deploy tool and orchestrator for our cloud deployments and strategy.

HAL Structure

- Organizations (AWS Accounts and Credentials)
- Applications (Terraform or Application Code Repos)
- Environments (TEST-AWS/BETA-AWS/PROD-AWS)
- Deployment Targets (Terraform Modules or Servers for Code Pushes)
- Builds (Terraform Plan or Actual Code Build)
- Pushes (Terraform Apply or Actual Code Push)
- Builds code, using Docker images, to various platforms

Controlled with a **.hal9000.yml** file. Highly configurable, steps, arbitrary commands, etc.
Tied into Active Directory allowing for deployment permission delegation and segmentation.

.hal9000.yml



- Primary HAL configuration file.
- Allows various platforms and images for building code.
- Standard steps include, build, before_deploy, after_deploy, etc.
- Environment variables to specify per-environment configuration
- Allow running of arbitrary scripts

Terraform version

Simple Example Terraform 11:

```
platform: 'linux'
image: 'iac'

build:  'plan-terraform-v2'
deploy: 'apply-terraform-v2'
```

https://docs.hal.foc.zone

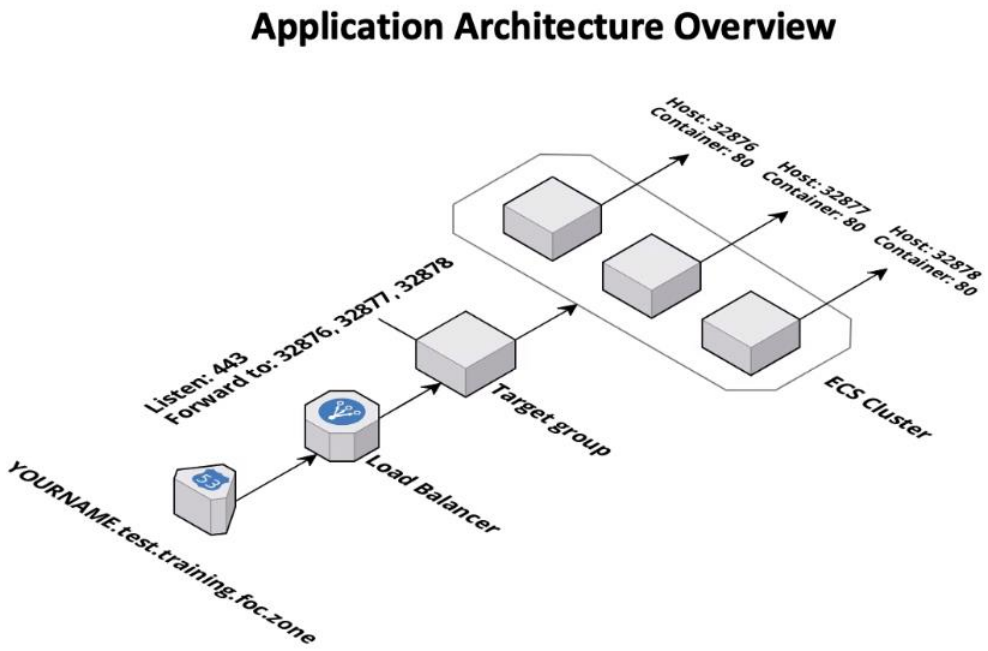
Terragrunt

Terragrunt is a thin wrapper for Terraform that provides extra tools for running terraform against a remote module.

Example of what you'll see:

```
terragrunt_source = "git::https://git.rockfin.com/terraform/aws-apigw-lambda-proxy-tf.git?ref=x.x.x"
```

What happened in Exercise 2

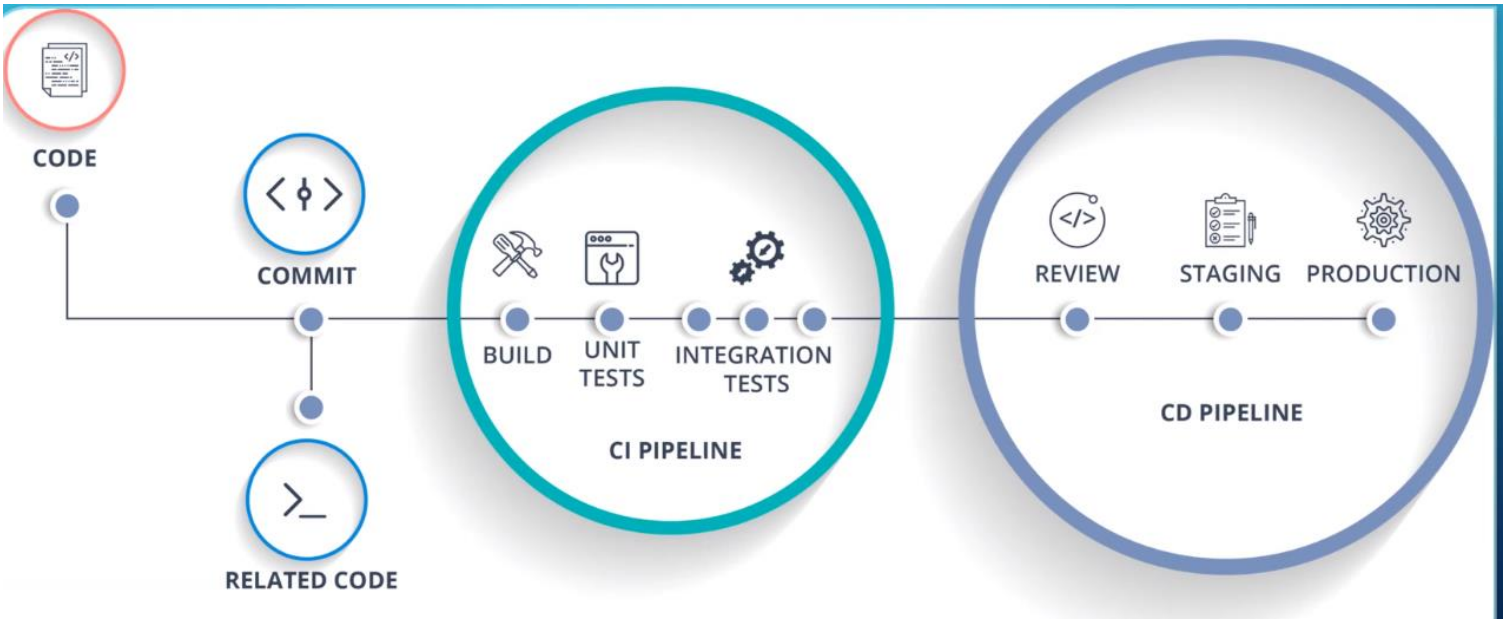


BUILDING A CI/CD PIPELINE WITH HAL & CIRCLECI

CONTINUOUS INTEGRATION / CONTINUOUS DEPLOYMENT

WHAT IS CI/CD?

- **Continuous Integration (CI)**
  - Automated build and testing of software.
  - Allow more regular merging (integration) of code branches
- **Continuous Deployment (CD)**
  - Automated deployment of built, tested code.
  - Once a build makes it through CI, it can be deployed to test or beta automatically.
- **Why CI/CD ?**
  - CI aims to speed up the release process by enabling teams to find and fix bugs earlier in the development cycle and encouraging stronger collaboration between developers – making it a crucial practice for agile teams.
  - Continuous delivery (CD) is the process of getting new builds into the hands of users as quickly as possible. It is the natural next step beyond CI and is an approach used to minimize the risks associated with releasing software and new features
  - Less human interaction
    - Less error prone
    - More time saved.
  - Deliver higher quality software.



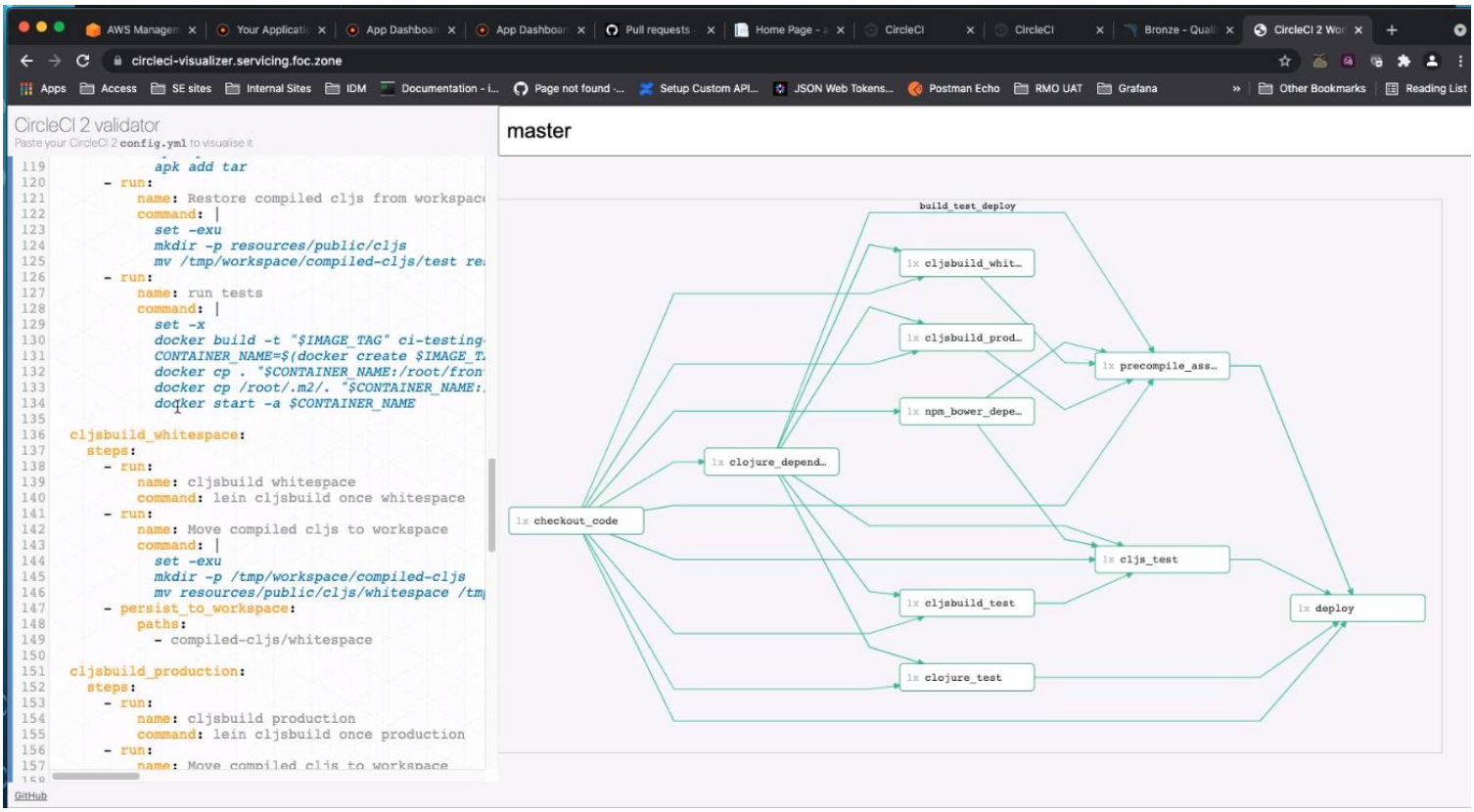
WHAT IS UNIT / INTEGRATION TESTING

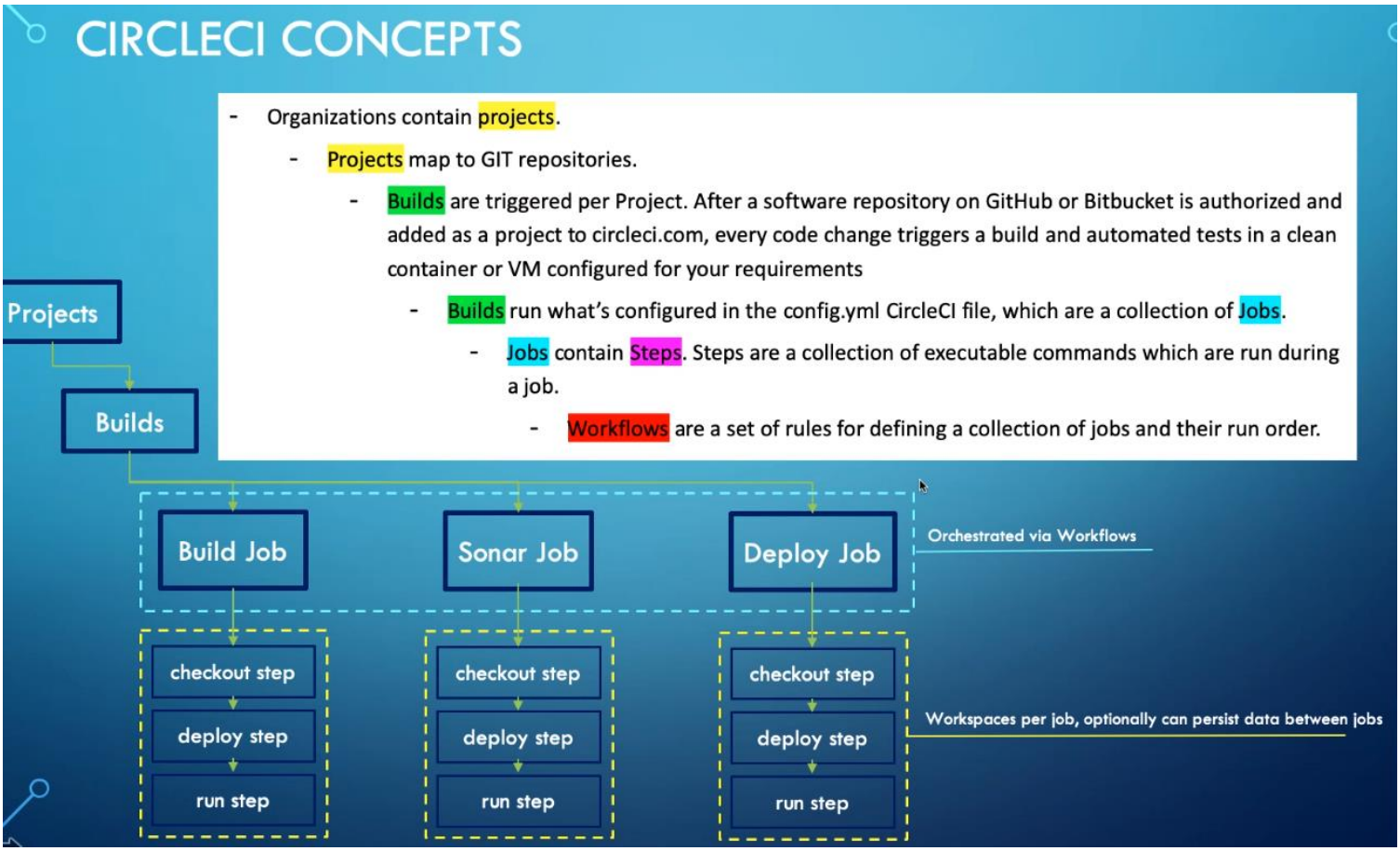
Unit Testing	Integration Testing
Unit testing is a type of testing to check if the small piece of code is doing what it is supposed to do.	Integration testing is a type of testing to check if different pieces of the modules are working together.
Unit tests should have no dependencies on code outside the unit tested.	Integration testing is dependent on other outside systems like databases, hardware allocated for them etc.
The goal of Unit testing is to test each unit separately and ensure that each unit is working as expected.	The goal of Integration testing is to test the combined modules together and ensure that every combined modules are working as expected.
It is usually written and executed by the developer.	It is usually executed by a test team, but can be done by a developer or automation scripts as well.

WHAT IS CIRCLECI?

- **CircleCI**
  - CircleCI is a Continuous Integration tool.
  - Structured as such:
    - Projects / Builds
    - Jobs / Steps / Workflows
  - After a software repository on GitHub is authorized and added as a project, every code change triggers automated tests in a clean container or VM.
  - CircleCI then sends an email notification of success or failure after the tests complete.
  - Almost completely controlled via a config.yml file contained within a .circleci folder which resides in a repo.

https://circleci-visualizer.servicing.foc.zone/





WHAT IS HAL?

- **What is HAL?**

  - QL Developed Code Deployment tool.
  - Historically Linux Server Deploy Tool
  - Has been used heavily in our transition to cloud deployments.

- **HAL Structure**

  - Organizations (AWS Accounts and Credentials)
  - Applications (Application Code Repos)
  - Environments (TEST-AWS / BETA-AWS / PROD-AWS)
  - Deployment Targets (Links environments to credentials)
  - Builds (Runs commands in .hal9000 file specified in BUILD phase)
  - Deploys (Runs commands in .hal9000 file specified in DEPLOY phase)
  - Builds code, using Docker images, to various platforms
  - Controlled with a **.hal9000.yml** file. Highly configurable, steps, arbitrary commands, etc.
  - Tied into Active Directory allowing for deployment permission delegation and segmentation.

https://DOCS.HAL.ZONE/API

WORKFLOW OF ECS CI/CD PIPELINE

1.) You commit and push a change in Git.

GitHub

2.) This triggers CircleCI to run the contents of its config.yml file, which for this demo means it will build your container, tag it to ECR, and push it there. It will then run the **publish_build_to_hal** step which gives it off to HAL.

circleci

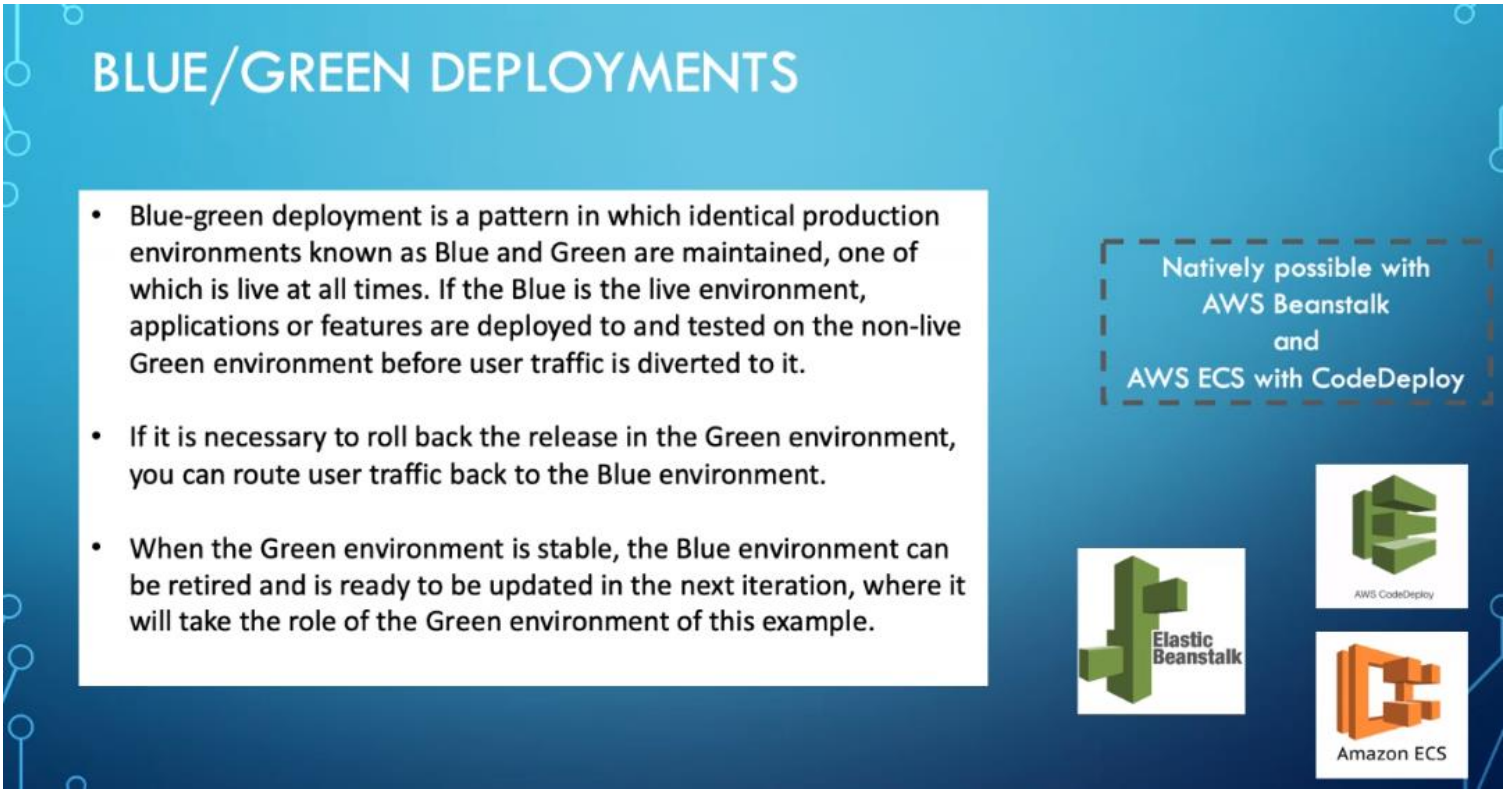
3.) **Publish_build_to_hal** runs a shell script on a docker container (*qldockerdtr.rockfin.com/circleci/hal-integration:hal-publisher*) which triggers a build and push in HAL. This causes the .hal9000.yml file to be read and executed, which does 2 things:

- Creates a new task definition, pointing to the new image that CircleCI just got done pushing to ECR.
- Updating the service on the cluster to use that new task definition.

Hal 2.14.1

4.) ECS received the update-service command from HAL and spins up *X* new containers linked to the new task definition. Once in the load balancer and healthy, it terminates the old containers.

docker



SECRET INJECTION – HOW TO HANDLE

- Most applications use and require secrets, whether that be a user/password, oauth client ID, splunk token, etc.
- These need to be injected securely at build time, run time, or stored in a secret store where they can be retrieved during container start-up.
- HAL, CircleCI, and AWS all support encrypted variables.
- Build time would be handled in CircleCI, run time would be handled in HAL.
- Secrets could also be stored in AWS Parameter Store or AWS Secrets Manager and retrieved via API call on container startup, or at a scheduled interval by application, etc.

SECRET INJECTION – BUILD TIME

Doing a find/replace on the Dockerfile with secrets stored in CircleCI, thus the container will be built with the secrets. (**build time**)

CircleCI config.yml

```
- run:
    name: Hydrate Secrets
    command: |
        sed -i -e "s~\${PUNISHER_TEST_BETA_NEXUS_CLIENT_ID}~$PUNISHER_TEST_BETA_NEXUS_CLIENT_ID~;" Dockerfile
        sed -i -e "s~\${PUNISHER_TEST_BETA_NEXUS_CLIENT_SECRET}~$PUNISHER_TEST_BETA_NEXUS_CLIENT_SECRET~;" Dockerfi
        sed -i -e "s~\${PUNISHER_PROD_NEXUS_CLIENT_ID}~$PUNISHER_PROD_NEXUS_CLIENT_ID~;" Dockerfile
```

Dockerfile

```
18  # Do the thing to run the thing
19  ADD . /webapp
20  ENV HOME /webapp
21  ENV PUNISHER_TEST_BETA_NEXUS_CLIENT_ID=${PUNISHER_TEST_BETA_NEXUS_CLIENT_ID}
22  ENV PUNISHER_TEST_BETA_NEXUS_CLIENT_SECRET=${PUNISHER_TEST_BETA_NEXUS_CLIENT_SECRET}
23  ENV PUNISHER_PROD_NEXUS_CLIENT_ID=${PUNISHER_PROD_NEXUS_CLIENT_ID}
```

SECRET INJECTION – RUN TIME

Secrets injected via HAL into the task.json so container starts up with them as environment variables. You configure them as encrypted configuration variables and then use a SED command in your .hal9000 file to replace variables in your task.json with those encrypted variables.

Variable Name	Scope	Value
ENCRYPTED_rds_password	test-aws	...73&#
ENCRYPTED_rds_password	beta-aws	...bfeh

```
deploy:

- sed -i -e "s~\${rds_password}~$encrypted_rds_password~;
```

.hal9000.yml file – rds_password in task.json replaced with encrypted_rds_password in HAL

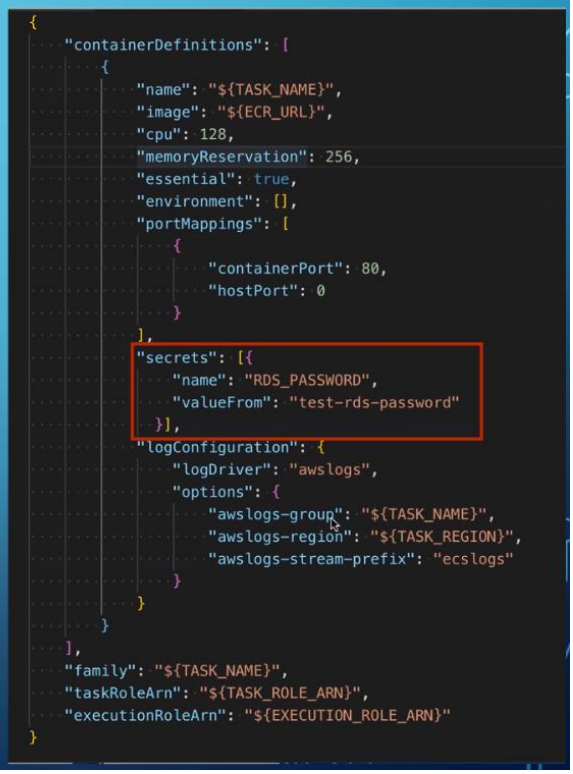
```
{
    "containerDefinitions": [
        {
            "name": "${TASK_NAME}",
            "image": "${ECR_URL}",
            "cpu": 128,
            "memoryReservation": 256,
            "essential": true,
            "environment": [
                "rds_password": "${rds_password}"
            ],
```

SECRET INJECTION – RUN TIME CONTINUED

AWS SSM Parameter Store & AWS Secrets Manager

- Native integration with ECS where secrets are injected via ECS task execution role into the task.json so container starts up with them as environment variables
- Preferred over injecting via HAL as encrypted configurations since they won't be stored in plaintext in the task definition.
- Can be encrypted with KMS and then only the IAM role with the correct permissions can decrypt.
- Parameter store is free and easy to use. Secrets Manager costs .40 cents per secret and offers more features and functionality, such as auto-rotating secrets for connecting to Aurora RDS.

https://docs.aws.amazon.com/AmazonECS/latest/developerguide/specifying-sensitive-data.html



Security TIP: AWS System Manager -> Parameter Store --- for storying secreats in AWS process, instead of storing in other repository and config files.

Create new Secret: AWS Secrets Manager -> Secret

Saves time of redeploying of code for any secret changes done/deleted