AWS Intensive

WELCOME!

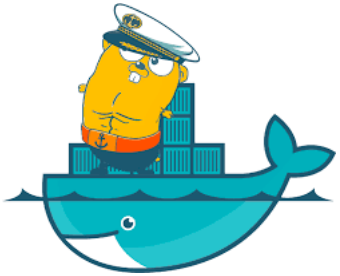
Develop Intelligence

Fernando Pombeiro

Overview of the AWS system

Our first Microservice tools foray

- So when we get to lab one we're going to be using **Docker-** which is a big part of the microservices architecture.
- We're using Docker because it's kind of a cheap way for us to get virtual machines without the overhead of installing something like Vagrant and iso images
- Think of Docker (in the context of this class) as a VM intended to get everyone onto the same OS.



Docker

- The easiest way to summarize docker is that you build the server around the code instead of moving the code into a server.
- This makes your server environments super easy to replicate, manage, and pass to your entire team.
- It also means (in the context of this class) that we have a way to make sure that all of us are on the same OS instead of worrying that half of us are following WINDOWS and the other half MAC.

The Dockerfile

- So the Dockerfile is sort of an all-in-one provisioning document that allows us to set up and execute a "VM" (I put that in quotes because big fans of docker insist that *it's not a vm!!* Even though I'm totally using it here as a VM).
- Anyways- Dockerfiles allow us to install all of the dependencies and packages in our "docker image" which is then instantiated as a "docker container" (which is totally a VM….fight me).

Docker Images and Containers

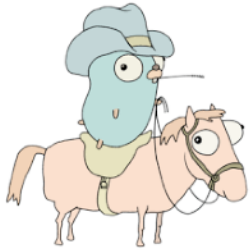
- Understanding the difference between images and containers is kind of the key to understanding DOCKER. If you've ever done Object Oriented Programming you'll understand this concept easily as IMAGES are the classes and CONTAINERS are the instantiated OBJECTS.

Docker Images and Containers (con't)

- So the idea here is that the Dockerfile builds an **image**. This image acts as the blueprint for the **container** that we build.
- In the broader context of microservices architecture DOCKER containers can act as small VMs that contain a single piece of code and execute it.
- We can have separate Docker containers for different functions in the app.

Final Note On Docker

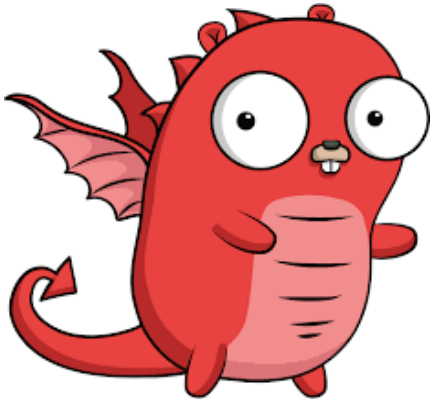
- A deep dive of Docker is outside of the scope of this class BUT…as we will be utilizing it extensively to ensure that everyone is on the same OS it's good to get a high level overview.
- If you don't use docker regularly in your microservices architecture then you're missing out. AWS has great services known as **Fargate and ECS** that act as repositories for docker containers. We'll get into these a bit later.



Infrastructure as code

- So it would be nice if we could list all of the assets that we need to provision, wire them all together, upload our code to them, and get everything running.
- Fortunately…thanks to the wonders of the modern technology…we CAN do *exactly* that.

- To do this we will be using **terraform**.



Introduction to Terraform

- So…think of terraform as a list of assets that you'll need in order to run your application. Like- you need:
  - Somewhere to store your data
  - Somewhere to present your data
  - Middleware
  - Code repository√

There are AWS services that basically do ALL of these things SO- with Terraform we list them and can run them.

BUT WAIT….there's more!!

Why Terraform?

- So there are several companies doing things similar to Terraform…including **serverless** and **cloudformation**…BUT- terraform has one big advantage:
  - It can work across platforms
  - So if a meteor takes out all of AWS you can move everything and move it to GCP without making a WHOLE lot of changes.
  - Also can work on Linode, Openstack, and Azure

So what does Terraform actually DO?

- So essentially you are able to manage most cloud platforms from your command line pretty easily (we'll be setting all of that up in the first lab shortly).
- You might use your CLI to say "create an EC2 server, an S3 bucket, a DynamoDB table, and a Lambda in the US-West-2 region". You enter those commands and then move on.
- Think of Terraform as the place you save those commands.

Terraform: More information

- So the other advantages of Terraform is that it not only **builds** your infrastructure but also changes and versions it appropriately.
- This gives you a lot of the same advantages you get from a coding repository- like rollbacks and diffs to let you see everything that's being changed.
- Think of things like **puppet, chef,** and **ansible** for provisioning servers.

Terraform (continued)

- Every Terraform process is broken down into **three separate and distinct parts**:
  - Plan
  - Apply
  - Execute

  We'll be getting further into these three later but for now suffice to say that this is the foundation for keeping your **infrastructure as code**

Users and Permissions

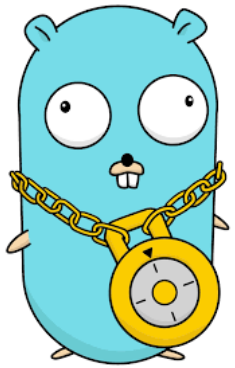
- In this first lab we're going to be setting up users and permissions…which will play a large part in your job as dev ops over any AWS environment.
- The first thing to understand about users is a similar rule to provisioning any sort of node; **avoid giving root access to people if possible**.
- Just like with nodes you provision or production/dev/staging nodes you have in existence: create users with permissions to do what they need to and then stop.

Users, Roles, Groups

- So understanding the **Identity and Access Management** system in **AWS** mostly concerns three concepts…which we'll be going over here:
  - **USERS:** This is the entity that you create in the IAM management console that represents an individual
  - **GROUPS:** This is a collection of IAM USERS. You would use GROUPS to grant/demove permissions for multiple users at the same time (think "HR" or "DEVS" or "ACCOUNTING")
  - **ROLES:** Roles are a set of permissions that can be assumed by **USERS** or **GROUPS** (of users) that give access to AWS functions.

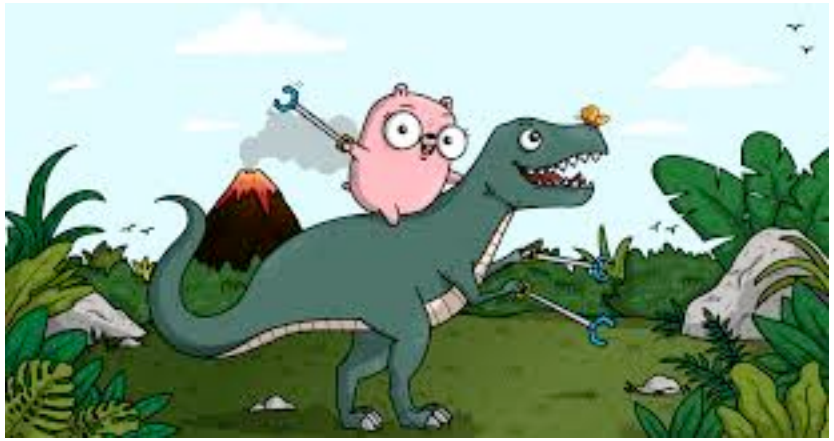
Users, Roles, Groups

- It's been my experience that when you have issues with something in your AWS architecture- be it nodes failing to communicate or something failing to put/pull data from a source it is either **permissions** or **VPC** related.
- It will be tempting to give max permissions and roles to every "user" that assumes a role to reduce errors. **Avoid the temptation to do this**.



Temporary Credentials

- With IAM you can also assign **temporary credentials** to an entity (or user) to “assume” a role in order to carry out an operation.
- You can set temporary credentials to expire after a set period of time allowing for better security around certain roles. We’ll go over some of this in the labs.



Using IAM roles to grant permissions

- So **ROLES** are nice because you aren't limited to **only** assigning them to **USERS**; you can also assign them to things like **EC2 instances.**
- As an example- if I have a **Lambda function** that reads data from an s3 bucket and puts that data into Redshift via a kinesis stream we can simply assign the lambda a ROLE that gives s3, kinesis, and Redshift permissions while it runs.

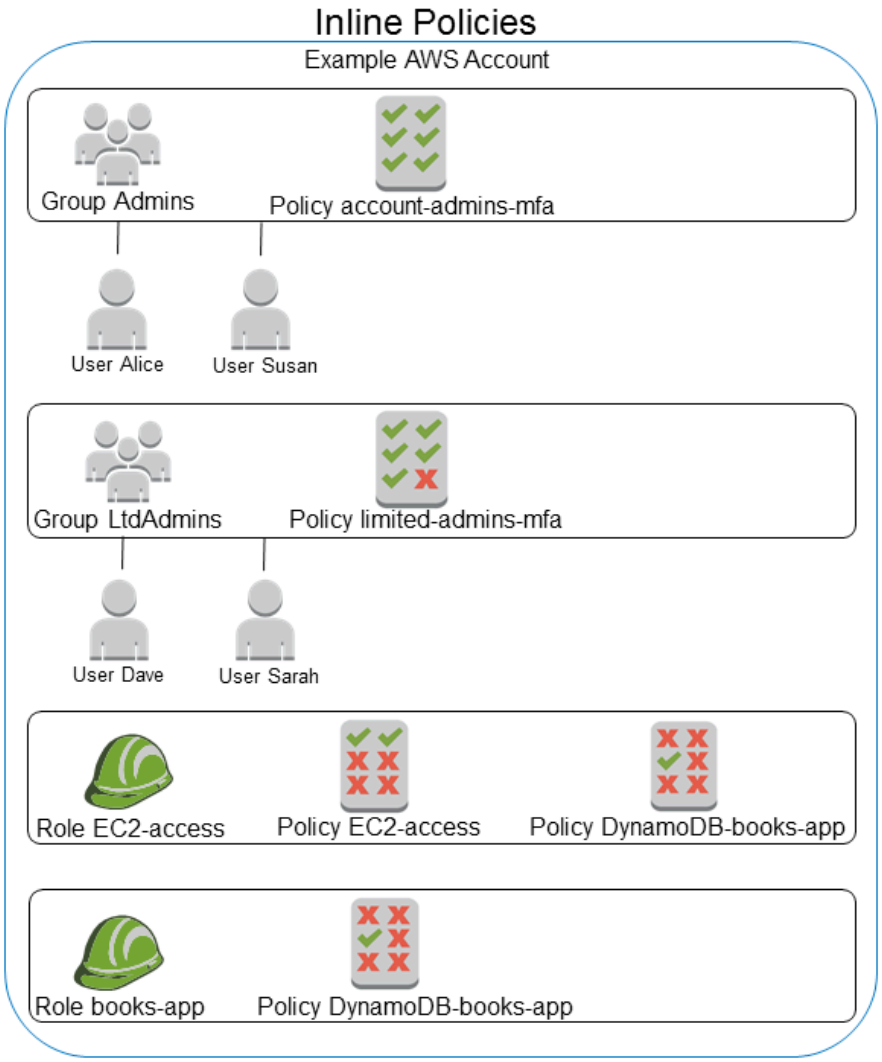
IAM setup (final thoughts)

- For this class we are going to use our AWS ROOT role to set up users to provision and run various parts of the architecture.
- The main thing that we'll need to get from creating the user is the **AWS_ACCESS_KEY** and **AWS_SECRET_ACCESS_KEY** for our users.
- Managing these keys is another challenge that we'll be going over in labs once docker and everything else is up and running.

Policies

- The final level to setting up users is to set **policies** which can be assumed by **ROLES.**
- **Policies define AWS permissions assigned to a user, group OR profile**. They are represented by collections of JSON that list the particular service and permissions attached to it.
- You can apply these policies in various ways- including granting limited access to certain AWS **resources** (like EC2 instances or a DynamoDB Table).

Policies (continued)



Policies

- Unique AWS resources can have their own policies attached to them and you can assign policies to a particular user to, for example, access EC2 instance 123 but **never** to access EC2 instance 456.
- Policies are a key component of allowing AWS resources to talk to each other…so for example you might have a Lambda function that wants to put things in an S3 bucket.
- In order to do this the **role** assumed by the Lambda function must have a **policy** that allows it to PUT into an s3 bucket.

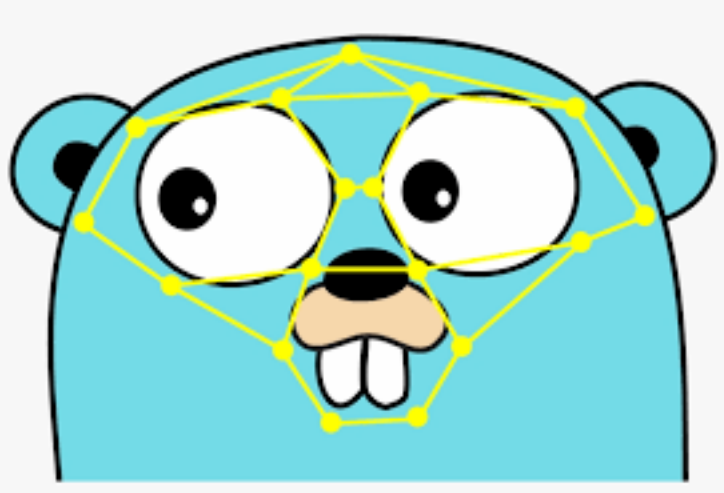
Policy documents in JSON

Different types of policies

- Identity Based Policies:
  - Basic policy- attached to IAM identities (users , groups and roles) that grant permissions to do things.
- Resource Based Policies:
  - The policies attached to resources– think of the s3 bucket example as that's a resource based policy.
- Permissions boundaries:
  - These are policies that limit permissions to a user or role…so "Jack can never have permission to PUT into an s3 bucket"/
- Organization Service Control Policies:
  - If you have a large organization you can set policies to define max permissions for all account belonging to an organizational unit (say accounting or HR)

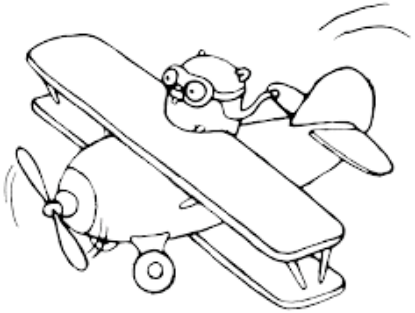
Different Types of Policies

- Access Control Lists
  - The only non-JSON based structures these are cross-accounts permissions policies that grant permissions to a principle entity.
- Session Policies:
  - “While logged in you can do X…but will expire at end of session”.



Describing AWS resources

- So now that we have been over IAM policies we need to address how AWS understands *resources.*
- Every resource in AWS is identified by an **ARN** or **Amazon Resource Name**.
- The ARN is an AWS service namespace that describes a resource unambiguously.
- Basically- it's AWS' name for all of the cool stuff that you are creating.



ARNs

- The basic format for an arn is as follows:

  arn:***partition***:***service***:*region*:*account-id*:*resourcetype*/*resource*

  And the various parts are made up as follows:

  - ***partition****:* unless you are in, like, China the partition will always be **aws**.
  - ***service:*** the aws service that we are using- for example S3, IAM, RDS
  - ***region:*** the standard aws regions (eu-west-1, eu-west-2, us-west-1, etc)
  - ***account:*** the account id of the account that owns the resource.
  - ***resource:*** usually the resource and a unique identifier

ARNS examples

- So, for example, for DynamoDB we might see an ARN that looks like this:

arn:aws:dynamodb:us-east-1:123456789012:table/books_table

- And for Lambda we might see something like this:

arn:aws:lambda:us-east-1:123456789012:function:ProcessKinesisRecords

We'll be using ARNs to describe stuff

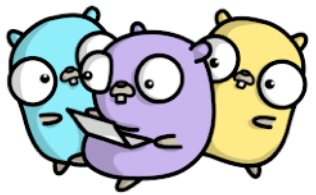
- ARNs are how we describe resources and anything we'd like to DO to those resources TO AWS via the command line
- ARNs can have specific policies attached to them (s3 bucket arns) and we can create policies for users that apply only to certain ARNs (Jack can access the ARN that controls our EC2 instance but does not have permissions to access any other ones)

AWS Secret Access Keys

- Each user will come with a set of keys- an ACCESS key and a SECRET key. We want to store these in places that are easily accessible and allow us to easily use the AWS command line.
- There are **several options** for key management with AWS. We're going to do **two** of them here- set them up locally (we'll do that in the lab) and set them up our Docker containers (as Environment variables).

Access Keys

- This part is important for security: ***do not put your secret or access keys in any sort of git repository***.
- Do not let your developers put keys anywhere public
- What does this mean? Basically…
  - If you are using keys locally from a file, add them to your .gitignore
  - If you need them as environment variables or to pass them as ARGs to a docker container…same rule read them or make them ENV variables on the host.
  - Do not ever, ever expose these.

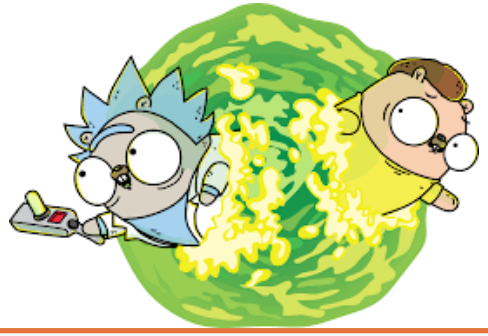


Final Note on Access Keys

- Access key management is one of those areas that there is no “right” answer for.
- We’re using Docker for this class and will be utilizing the “make your access keys environment variables” technique to manage our keys…but this is far from the only option available to us.
- Credentials files and profiles are also incredibly effective for key management.

Access Keys (best practices)

- My suggestion would be that you, as the devops team, keep copies of every set of access keys that you grant alongside of every user ID. This can come in handy if you have to update the keys (should be obvious how to do this) and if a user loses them.
- Upon a developer leaving you should remember to go to the IAM console and remove their access as soon as possible- including the keys.



Confused????

- GOOD! Ask Questions!!