**IAM-Accounts-AWS-Organizations**

**IAM Identity Policies**

Identity Policies are attached to AWS Identities which are IAM users, IAM groups, and IAM roles. These are a set of security statements that ALLOW or DENY access to AWS resources.

When an identity attempts to access AWS resources, that identity needs to prove who it is to AWS, a process known as **Authentication**. Once authenticated, that identity is known as an **authenticated identity**

**Statement Components**

* **Statement ID (SID):** Optional field that should help describe
  + The resource you're interacting
  + The actions you're trying to perform
* **Effect:** is either allow or deny.
  + It is possible to be allowed and denied at the same time
* **Action** are formatted service:operation. There are three options:
  + specific individual action
  + wildcard as an action
  + list of multiple independent actions
* **Resource:** similar to action except for format arn:aws:s3:::catgifs

**Priority Level**

* **Explicit Deny:** Denies access to a particular resource cannot be overruled.
* **Explicit Allow:** Allows access so long there is not an explicit deny.
* **Default Deny (Implicit):** IAM identities start off with no resource access.

**Inline Policies and Managed Policies**

* **Inline Policy:** grants access and assigned on each accounts individually.
* **Managed Policy (best practice):** one policy is applied to all users at once.

**IAM Users**

Identity used for anything requiring **long-term** AWS access

* Humans
* Applications
* Service Accounts

If you can name a thing to use the AWS account, this is an IAM user.

When a **principal** wants to **request** to perform an action, it will **authenticate** against an identity within IAM. An IAM user is an identity which can be used in this way.

There are two ways to authenticate:

* **Username and Password**
* **Access Keys (CLI)**

Once the **Principal** has authenticated, it becomes an **authenticated identity**

**Amazon Resource Name (ARN)**

Uniquely identify resources within any AWS accounts.

This allows you to refer to a single or group of resources. This prevents individual resources from the same account but in different regions from being confused.

ARN generally follows the same format:

arn:partition:service:region:account-id:resource-id

arn:partition:service:region:account-id:resource-type/resource-id

arn:partition:service:region:account-id:resource-type:resource-id

* **partition:** almost always aws unless it is china aws-cn
* **region:** can be a double colon (::) if that doesn't matter
* **account-id:** the account that owns the resource
  + EC2 needs this
  + S3 does not need account-id because its globally unique
* **resource-type/id:** changes based on the resource

An example that leads to confusion:

* arn:aws:s3:::catgifs
  + This references an actual bucket
* arn:aws:s3:::catgifs/\*
  + This refers to objects in that bucket, but not the bucket itself.

These two ARNs do not overlap

**IAM FACTS**

* 5,000 IAM users per account
* IAM user can be a member of 10 groups

**IAM Groups**

Containers for users. **You cannot login to IAM groups** They have no credentials of their own. Used solely for management of IAM users.

Groups bring two benefits

1. Effective administrative style management of users based on the team
2. Groups can have Inline and Managed policies attached.

AWS merges all of the policies from all groups the user is in together.

* The 5000 IAM user limit applies to groups.
* There is **no all users** IAM group.
  + You can create a group and add all users into that group, but it needs to be created and managed on your own.
* No Nesting: You cannot have groups within groups.
* 300 Group Limit per account. This can be fixed with a support ticket.

**Resource Policy** A bucket can have a policy associated with that bucket. It does so by referencing the identity using an ARN (Amazon Reference Name). A policy on a resource can reference IAM users and IAM roles by the ARN. A bucket can give access to one or more users or one or more roles.

**GROUPS ARE NOT A TRUE IDENTITY** **THEY CAN'T BE REFERENCED AS A PRINCIPAL IN A POLICY**

An S3 Resource cannot grant access to a group, it is not an identity. Groups are used to allow permissions to be assigned to IAM users.

**IAM Roles**

A single thing that uses an identity is an IAM User.

IAM Roles are also identities that are used by large groups of individuals. If have more than 5000 principals, it could be a candidate for an IAM Role.

IAM Roles are **assumed** you become that role.

This can be used short term by other identities.

IAM Users can have inline or managed policies which control which permissions the identity gets within AWS.Policies which grant, allow or deny, permissions based on their associations.

IAM Roles have two types of policies can be attached.

* **Trust Policy:** Specifies which identities are allowed to assume the role.
* **Permissions Policy:** Specifies what the role is allowed to do.

If an identity is allowed on the **Trust Policy**, it is given a set of **Temporary Security Credentials**. Similar to access keys except they are time limited to expire. The identity will need to renew them by reassuming the role.

Every time the **Temporary Security Credentials** are used, the access is checked against the **Permissions Policy**. If you change the policy, the permissions of the temp credentials also change.

Roles are real identities and can be referenced within resource policies.

**Secure Token Service** (sts:AssumeRole) this is what generates the temporary security credentials (TSC).

**When to use IAM Roles**

**Lambda Execution Role.** For a given lambda function, you cannot determine the number of principals which suggested a Role might be the ideal identity to use.

* **Trust Policy:** to trust the Lambda Service
* **Permission Policy:** to grant access to AWS services.

When this is run, it uses the sts:AssumeRole to generate keys to CloudWatch and S3.

It is better when possible to use an IAM Role versus attaching a policy.

**Emergency or out of the usual situations**

Break Glass Situation - There is a key for something the team does not normally have access to. When you break the glass, you must have a reason to do. A role can have an Emergency Role which will allow further access if it’s really needed.

**Adding AWS into existing corporate environment**

You may have an existing identity provider you are trying to allow access to. This may offer SSO (Single Sign On) or over 5000 identities. This is useful to reuse your existing identities for AWS. External accounts can't be used to access AWS directly. To solve this, you allow an IAM role in the AWS account to be assumed by one of the active directories. **ID Federation** allowing an external service the ability to assume a role.

**Making an app with 1,000,000 users**

**Web Identity Federation** uses IAM roles to allow broader access. These allow you to use an existing web identity such as google, facebook, or twitter to grant access to the app. We can trust these web identities and allow those identities to assume an IAM role to access web resources such as DynamoDB. No AWS Credentials are stored on the application. Can scale quickly and beyond.

**Cross Account Access**

You can use a role in the partner account and use that to upload objects to AWS resources.

**AWS Organizations**

Without an organization, each AWS account needs its own set of IAM users as well as individual payment methods. If you have more than 5 to 10 accounts, you would want to use an org.

Take a single AWS account **standard AWS account** and create an org. The standard AWS account then becomes the **master account**. The master account can invite other existing standard AWS accounts. They will need to approve their joining to the org.

When standard AWS accounts become part of the org, they become **member accounts**. Organizations can only have one **master accounts** and zero or more **member accounts**

**Organization Root**

This is a container that can hold AWS member accounts or the master account. It could also contain **organizational units** which can contain other units or member accounts.

**Consolidated billing**

The individual billing for the member accounts is removed and they pass their billing to the master account. Inside an AWS organization, you get a single monthly bill for the master account which covers all the billing for each users. Can offer a discount with consolidation of reservations and volume discounts

**Create new accounts in an org**

Adding accounts in an organization is easy with only an email needed. You no longer need IAM users in each accounts. You can use IAM roles to change these. It is best to have a single AWS account only used for login. Some enterprises may use an AWS account while smaller ones may use the master.

**Role Switching**

Allows you to switch between accounts from the command line

**Service Control Policies**

Can be used to restrict what member accounts in an org can do.

JSON policy document that can be attached:

* To the org as a whole by attaching to the root container.
* A specific Organizational Unit
* A specific member only.

The master account cannot be restricted by SCPs which means this should not be used because it is a security risk.

SCPs limit what the account, **including root** can do inside that account. They don't grant permissions themselves, just act as a barrier.

**Allow List vs Deny List**

Deny list is the default.

When you enable SCP on your org, AWS applies FullAWSAccess. This means SCPs have no effect because nothing is restricted. It has zero influence by themselves.

{

"Version": "2012-10-17",

"Statement": {

"Effect": "Allow",

"Action": "\*",

"Resource": "\*"

}

}

SCPs by themselves don't grant permissions. When SCPs are enabled, there is an implicit deny.

You must then add any services you want to Deny such as DenyS3

{

"Version": "2012-10-17",

"Statement": {

"Effect": "Deny",

"Action": "s3:\*",

"Resource": "\*"

}

}

**Deny List** is a good default because it allows for the use of growing services offered by AWS. A lot less admin overhead.

**Allow List** allows you to be conscience of your costs.

* To begin, you must remove the FullAWSAccess list
* Then, specify which services need to be allowed access.
* Example AllowS3EC2 is below

{

"Version": "2012-10-17",

"Statement": [

{

"Effect": "Allow",

"Action": [

"s3:\*",

"ec2:\*"

],

"Resource": "\*"

}

]

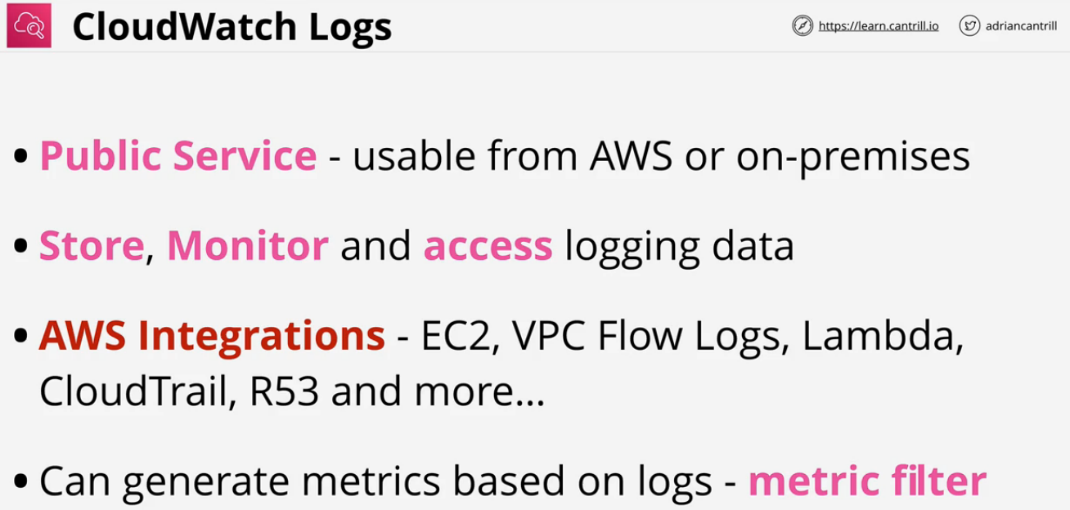
}

**CloudWatch Logs**

This is a public service, this can be used from AWS VPC or on premise environment.

This allows to **store**, **monitor** and **access** logging data.

* This is a piece of **information data** and a **timestamp**
* Can be more fields, but at least these two

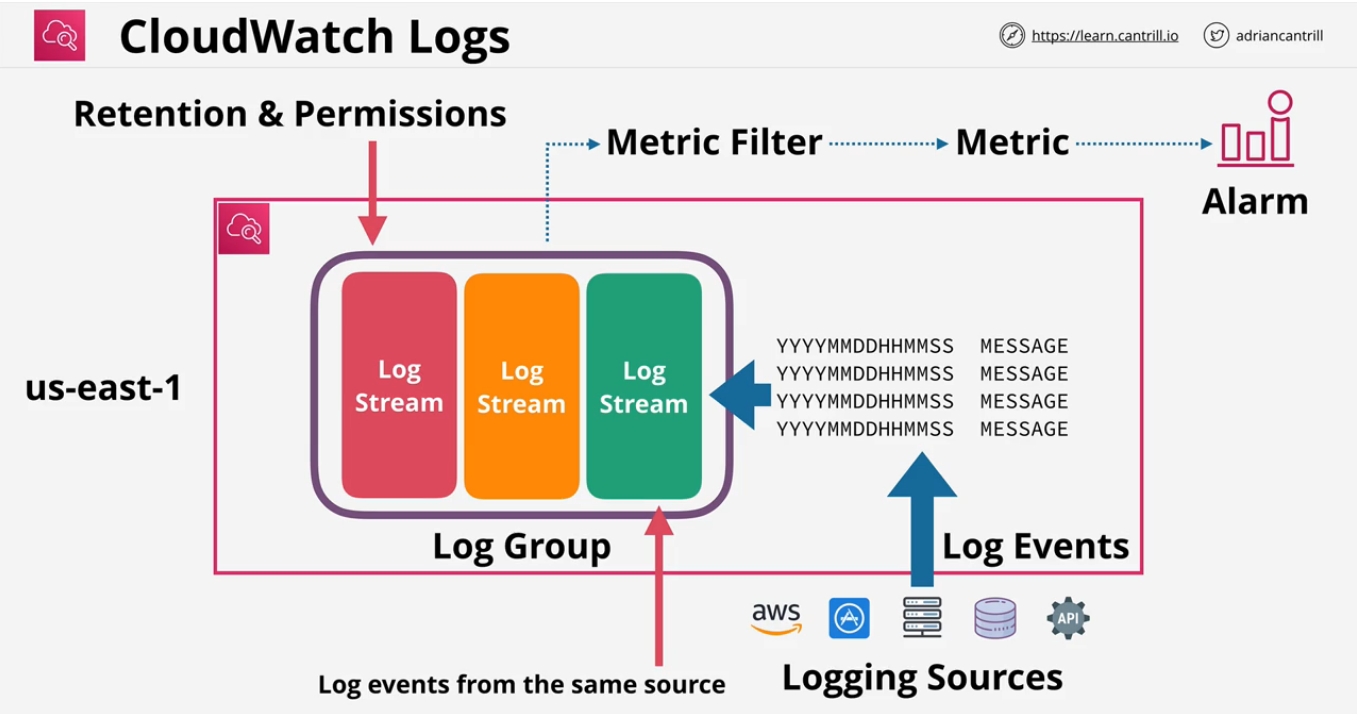
Comes with some **AWS Integrations**.

Security is provided with **IAM roles** or **Service roles** Can generate metrics based on logs **metric filter**

**Architecture of CloudWatch Logs**

It is a regional service us-east-1

Need **logging sources** such as AWS Product & Services, external APIs, servers or databases. This sends information as **log events**. These are stored in **log streams**. This is a sequence of log events from the same source.



**Log Groups** are containers for multiple logs streams of the same type of logging. This also stores **configuration settings** such as **retention settings** and **permissions.**

Once the settings are defined on a **log group**, they apply to all log streams in that log group. **Metric filters** are also applied on the log groups, which can be then used to configure alarms..

**CloudTrail Essentials**

Concerned with who did what.

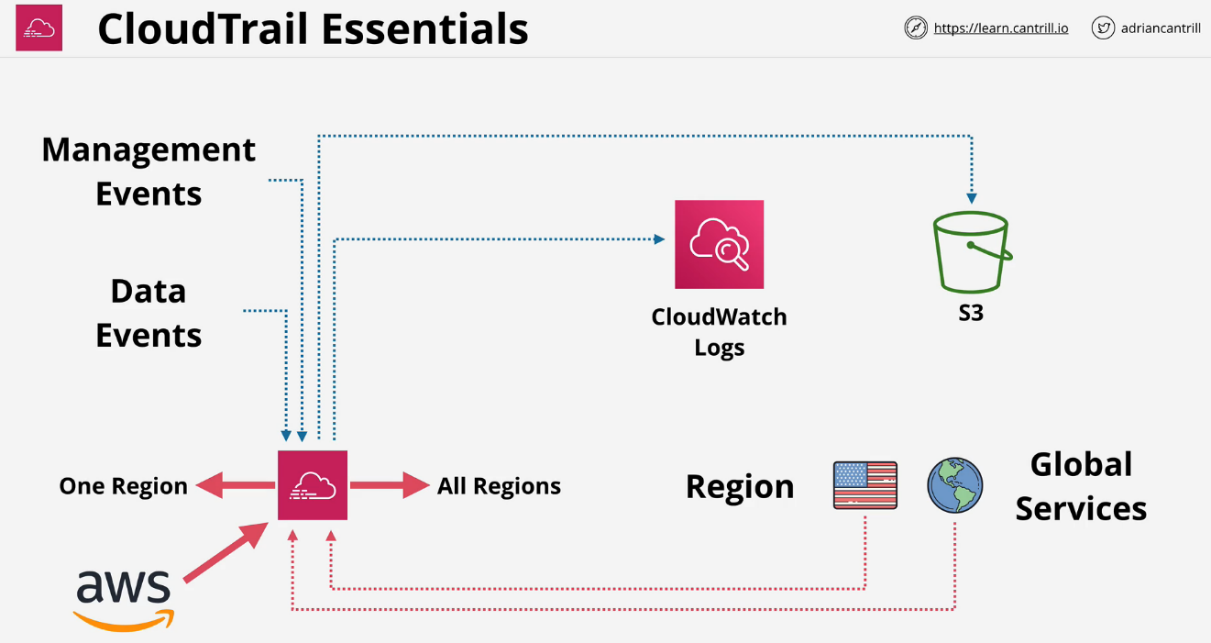
Logs API calls or activities as **CloudTrail Event**

Stores the last **90** days of events in the **Event History**. This is enabled by default and is no additional cost.

To customize the service you need to create a new **trail**. Two types of events. By **Default** only logs **Management Events**

* **Management Events:** Provide information about management operations performed on resources in the AWS account. Create an EC2 instance or terminating one.
* **Data Events:** Objects being uploaded to S3 or a Lambda function being invoked. This is not enabled by default and must be enabled for that trail.

**CloudTrail Trail**

Logs events for the AWS region it is created in. It is a regional service.

Once created, it can operate in two ways

* **One region trail**
* **All region trail**
  + Collection of trails in all regions
  + When new regions are added, they will be added to this trail automatically

Most services log events in the region they occur. The trail then must be a **one region trail** in that region or an **all-region trail** to log that event.

A small number of services log events globally to one region. **Global services** such as IAM or STS or CloudFront always log their events to us-east-1

A trail must have **Global Services Event Logging** enabled to have this logged.

AWS services are largely split into regional services or global services.

When the services log, they log in the region they are created or to us-east-1 if they are a global service.

A trail can store events in an S3 bucket as a compressed JSON file. It can also use CloudWatch Logs to output the data.

CloudTrail products can create an **organizational trail**. This allows a **single management point for all the APIs** and **management events for that org**.

**CloudTrail Exam PowerUp**

* It is enabled by default for 90 days without S3
* Trails are how you configure S3 and CWLogs
* Only Management events are saved by default
* IAM, STS, CloudFront are **Global Service** events and log to us-east-1
  + Trail must be enabled to do this
* **NOT REALTIME** - There is a delay. Approximately **15 minute** delay

**CloudTrail Pricing**

<https://aws.amazon.com/cloudtrail/pricing/>