AWS Organizations

- Standard AWS account: it is an account which is not in an AWS Organization
- We create an AWS Organization from a standard AWS account
- The organization is not created in this account, we just use the account to create the organization. The standard account then becomes the **Management Account** (used to be called *Master Account*)
- Using the Management Account we can invite other accounts into the organization
- When a standard account joins an organization, it will change to **Member Account** of that organization
- Organizations have 1 Management Account and 0 or more Member Accounts
- We can create a structure of AWS accounts in an organization. We can group accounts by things such as business units, function or development stage, etc.
- This structure is hierarchical, it is an inverted tree
- At the top of this tree is the root container of the organization (just a container within the organization, NOT to be confused with the root user)
- This root container can contain other containers, this containers are known as **Organizational Units (OU)**
- OUs can contains accounts (Management/Member accounts) or other OUs

Consolidated Billing

- It is an important feature of AWS Organizations
- The individual billing method of each account from the organization is removed, the member accounts pass their billing through the Management Account (**Payer Account**)
- Using consolidated billing we get a single monthly bill. This covers the Management Account and all the Member Accounts of the Organization
- When using organization reservation benefits and discounts are pooled, meaning the organization can benefit as a whole for the spending of each AWS account within the org

Best Practices

- Have a single account into which users can log into and assume IAM roles in order to access other accounts from the org
- The account with all the identities may be the Management Account or it can be another Member Account (*Login Account*)

`OrganizationAccountAccessRole`

- This is an IAM role used to access the newly added/created account in an organization
- This role will be created automatically if we create the account from an existing organization
- This role has to be created manually in the member account if the account was invited into the organization

Service Control Policies (SCP)

- They are a feature of AWS Organizations used to restrict AWS accounts
- They are JSON documents
- They can be attached to the root of the organization, to one or more OUs or to individual AWS accounts
- SCPs inherit down through the organization tree
- The Management Account is special: even if it has SCPs attached (directly or through an OU) it wont be affected by the SCP
- SCPs are account permission boundaries:
  - They limit what the account (including the root user of the account) can do
  - We can never restrict a root user from an account, but we can restrict the account itself, hence these restrictions will apply to the root user as well
- **SCPs don't grant any permissions!** This are just a boundary to limit what is and is not allowed in an account
- SCPs can be used in two ways:
  - Deny list (default): allow by default and block access to certain services
    - `FullAWSAccess`: policy applied by default to the org an all OUs when we enable SCPs. This policy means tha by default nothing is restricted
    - SCPs don't grant permissions, but when they are enabled, there is a default deny for everything. This is why the `FullAWSAccess` policy is needed
    - SCP priority rules:
      1. Explicit Deny
      2. Allow
      3. Default (implicit) deny
    - Benefits of deny lists is that as AWS is extends the list of service offerings, new services will be available for accounts (low admin overhead)
  - Allow list: block by default and allow certain services
    - To implement allow lists:
      1. Remove the `FullAWSAccess` policy

2. Add any services which should be allowed in a new policy

∗ Allow lists are more secure, but they require more admin overhead

AWS Resource Access Manager - RAM

- Allows sharing resources between AWS accounts
- Some services may allow sharing between any AWS accounts, some allow sharing only between accounts from the same organization
- Services needs to support RAM in order to be shared (not everything can be shared)
- Services can be shared with principals: accounts, OU's and ORG
- Shared resources can be accessed natively
- There is no cost by using RAM, only the service cost may apply
- AWS RAM for sharing resources in an organization can be enabled with `enable-sharing-with-aws-organizations` CLI command. This operation creates a service-linked role called `AWSServiceRoleForResourceAccessManager` that has the IAM managed policy named `AWSResourceAccessManagerServiceRolePolicy` attached. This role permits RAM to retrieve information about the organization and its structure. This lets us share resources with all of the accounts

Availability Zone IDs

- A region in AWS has multiple availability zones, example: `us-east-1a`, `us-east-1b`, etc.
- AWS rotates the name of the AZs depending on the AWS account, meaning that `us-east-1a` may not be the same AZ if we compare 2 accounts
- If a failure happens on the hardware level, two accounts may see the issue being in different AZ, this may introduce a challenge in troubleshooting
- AWS provides AZ IDs to overcome this challenge. Example of IDs: `use1-az1`, `use1-az2`
- AZ IDs are consistent across multiple accounts

RAM Concepts

- **Owner account**:
  - Owns the resource, creates a share, provides the name
  - Retains full permission over the resource shared
  - Defines the principal (AWS account, OU, entire AWS organization) with whom the share a specific resource
- **Principle**:
  - It can be an AWS account, OU, entire AWS organization
  - Resources are shared with a principle
- If the participant is inside an ORG with the sharing enabled, sharing is accepted automatically

- For non ORG accounts, or sharing with AWS Organizations is not enabled, we have to accept an invite

Shared Services VPC

- It is a VPC which provides infrastructure which can be used by other services
- In AWS this has been traditionally architected using separate networks connected using VPC peering or Transit Gateways. With AWS RAM and AWS Organizations we can create something which is more effective: Shared Services VPC
- VPC owner can create and manage the VPC and subnets which shared with participants
- Participants can provision services into the shared subnets, can read an reference network objects but can not modify or delete the subnets
- Resources created by a participant account will not be visible for other participants or by the VPC owner account
- Resources created by a participant account can be accessed from other resources created by other participant accounts because they are on the same network

Policies

- IAM policies define permissions for an action regardless of the method that you use to perform the operation

Policy types

- **Identity-based policies**: attach managed and inline policies to IAM identities (users, groups to which users belong, or roles). Identity-based policies grant permissions to an identity
- **Resource-based policies**: attach inline policies to resources. The most common examples of resource-based policies are Amazon S3 bucket policies and IAM role trust policies. Resource-based policies grant permissions to a principal entity that is specified in the policy. Principals can be in the same account as the resource or in other accounts
- **Permissions boundaries**: use a managed policy as the permissions boundary for an IAM entity (user or role). That policy defines the maximum permissions that the identity-based policies can grant to an entity, but does not grant permissions. Permissions boundaries do not define the maximum permissions that a resource-based policy can grant to an entity
- **Organizations SCPs**: use an AWS Organizations service control policy (SCP) to define the maximum permissions for account members of an organization or organizational unit (OU). SCPs limit permissions that identity-based policies or resource-based policies grant to entities (users or

roles) within the account, but do not grant permissions

- **Access control lists (ACLs)**: use ACLs to control which principals in other accounts can access the resource to which the ACL is attached. ACLs are similar to resource-based policies, although they are the only policy type that does not use the JSON policy document structure. ACLs are cross-account permissions policies that grant permissions to the specified principal entity. ACLs cannot grant permissions to entities within the same account
- **Session policies**: pass advanced session policies when you use the AWS CLI or AWS API to assume a role or a federated user. Session policies limit the permissions that the role or user's identity-based policies grant to the session. Session policies limit permissions for a created session, but do not grant permissions. For more information, see Session Policies

Policies Deep Dive

- Anatomy of a policy: JSON document with `Effect`, `Action`, `NotAction` (inverse condition of `Action`), `Resource`, `Conditions` and `Policy Variables`
- Priority order of permissions in AWS is: deny (explicit) > allow > deny (implicit). A policy always assumes a default (implicit) deny => if we do not allow explicitly to do something, we wont be able to do it
- An explicit `DENY` has always precedence over `ALLOW`
- Best practice: use least privilege for maximum security
  - Access Advisor: a tool for seeing permissions granted and when last accessed
  - Access Analyzer: used of analyze resources shared with external entities
- Common Managed Policies:
  - `AdministratorAccess`
  - `PowerUserAccess`: does not allow anything regarding to IAM, organizations and account (with some exceptions), otherwise similar to admin access
- IAM policy condition:

```
"Condition": {
    "{condition-operator}": {
        "{condition-key}": "{condition-value}"
    }
}
```

- Operators:
  - String: `StringEquals`, `StringNotEquals`, `StringLike`, etc.

    - Numeric: `NumericEquals`, `NumericNotEquals`, `NumericLessThan`, etc.
    - Date: `DateEquals`, `DateNotEquals`, `DateLessThan`, etc.
    - Boolean
    - IpAddress/NotIpAddress:
        - `"Condition": {"IpAddress": {"aws:SourceIp": "192.168.0.1/16"}}`
    - ArnEquals/ArnLike
    - Null
        - `"Condition": {"Null": {"aws:TokenIssueTime": "192.168.0.1/16"}}`
- Policy Variables and Tags:
    - `${aws:username}`: example `"Resource:["arn:aws:s3:::mybucket/${aws:username}/*"]`
    - AWS Specific:
        - `aws:CurrentTime`
        - `aws:TokenIssueTime`
        - `aws:PrincipalType`: indicates if the principal is an account, user, federated or assumed role
        - `aws:SecureTransport`
        - `aws:SourceIp`
        - `aws:UserId`
    - Service Specific:
        - `ec2:SourceInstanceARN`
        - `s3:prefix`
        - `s3:max-keys`
        - `sns:EndPoint`
        - `sns:Protocol`
    - Tag Based:
        - `iam:ResourceTag/key-name`
        - `iam:PrincipalTag/key-name`

Permission Boundaries

- Only IDENTITY permissions are impacted by boundaries - any resource policies are applied full
- Permission boundaries can be applied to IAM Users and IAM Roles
- Permission boundaries don't grant access to any action. They define maximum permissions an identity can receive
- Use cases for permission boundaries:
    - Delegation problem: if we give elevated permissions to an user, he/she could promote itself to have administrator permissions or could create another user with administrator permissions
    - Solution is to have a boundary which forbids changing its onw user's permissions and forbid creating other users/roles with elevated permissions

Policy Evaluation Logic

- Components involved in a policy evaluations:
  - Organization SCPs
  - Resource Policies
  - IAM Identity Boundaries
  - Session Policies
  - Identity Policies
- Policy evaluation logic - same account: policy evaluation logic - same account
- Policy evaluation logic - different account: policy evaluation logic - different account

AWS Policy Simulator

- When creating new custom policies you can test it here:
  - https://policysim.aws.amazon.com/home/index.jsp
  - This policy tool can you save you time in case your custom policy statement's permission is denied
- Alternatively, you can use the CLI:
  - Some AWS CLI commands (not all) contain `--dry-run` option to simulate API calls. This can be used to test permissions.
  - If the command is successful, you'll get the message: `Request would have succeeded, but DryRun flag is set`
  - Otherwise, you'll be getting the message: `An error occurred (UnauthorizedOperation) when calling the {policy_name} operation`

IAM: Identity and Access Management

- When accessing AWS, the root account should **never** be used. Users must be created with the proper permissions. IAM is central to AWS
- **Users**: A physical person
- **Groups**: Functions (admin, devops) Teams (engineering, design) which contain a group of users
- **Roles**: Internal usage within AWS resources
  - **Cross Account Roles**: roles used to assumed by another AWS account in order to have access to some resources in our account
- **Policies (JSON documents)**: Defines what each of the above can and cannot do. **Note**: IAM has predefined managed policies
  - There are 3 types of policies:
    - * AWS Managed
    - * Customer Managed
    - * Inline Policies
- **Resource Based Policies**: policies attached to AWS services such as S3, SQS

IAM Roles vs Resource Based Policies

- When we assume a role (user, application or service), we give up our original permission and take the permission assigned to the role
- When using a resource based policy, principal does not have to give up any permissions
- Example: user in account A needs to scan a DynamoDB table in account A and dump it in an S3 bucket in account B. In this case if we assume a role in account B, we wont be able to scan the table in account A

Best practices

- One IAM User per person **ONLY**
- One IAM Role per Application
- IAM credentials should **NEVER** be shared
- Never write IAM credentials in your code. **EVER**
- Never use the ROOT account except for initial setup
- It's best to give users the minimal amount of permissions to perform their job

STS

- Allows to assume roles across different accounts or same accounts
- Generates temporary credentials (`sts:AssumeRole*`)
- Temporary credentials are similar to access key. They expire and they don't directly belong to the identity which assumes the role
- Temporary credentials usually provide limited access
- Temporary credentials are requested by another identity (AWS or external - identity federation)
- Temporary credentials include the following:
  - `AccessKeyId`: unique ID of the credentials
  - `Expiration`: date and time of credential expiration
  - `SecretAccessKey`: used to sign the requests to AWS
  - `SessionToken`: unique token which must be passed with all the requests to AWS
- STS allows us to enable identity federation

Assume a Role with STS

1. Define an IAM role within an account or cross-account
2. Define which principals can access the IAM role
3. Use the AWS STS (Secure Token Service) to retrieve the IAM role we have access to (`AssumeRole` API)
4. Temporary credentials can be valid between 15 minutes to 1 hour

Revoke IAM Role Temporary Credentials

- **Trust policy**: specifies who can assume a role
- Roles can be assumed by many identities
- Everybody who assumes a role, gets the same set of permissions
- Temporary credentials can not be cancelled, they are valid until they expire
- Temporary credentials can last for longer time
- In case of a credential leak if we change the permissions for the policy, we will affect all legitimate users - not a good idea for revoking access
- Solution:
  - Revoke all existing sessions, by applying an `AWSRevokeOlderSessions` inline policy to the role. This will apply to all existing sessions, sessions created afterwards will not be affected
  - We can not manually revoke credentials!

Multi-Factor Authentication (MFA)

- **Factor**: different piece of evidence which proves the identity
- Factors:
  - **Knowledge**: something we as users know: username, password
  - **Possession**: something we as users have: bank card, MFA device/app
  - **Inherent**: something we are, example: fingerprint, face, voice, iris
  - **Location**: a location (physical) or which network we are connected to (corporate wifi)
- More factors means more security, harder to bypass by an intruder

AWS Service Quotas

- Defines how much of a "thing" we can use inside of an AWS account
- Example:
  - Number of EC2 instances at a certain times per region
  - Number of IAM users per AWS accounts
- Services usually have a default per region quota
- Global services may have a per account quota instead per region
- Most services quotas can be increased as needed
- Some service quotes can not be changed, example: number of IAM users per account (5000)
- Service endpoint and quotas: https://docs.aws.amazon.com/general/latest/gr/aws-service-information.html
- **Service Quotas**:
  - From the console we can go to *Service Quotas* page, where we can create dashboards for quotas we want to monitor
  - We can request quota changes from this service for certain services
  - *Quote request template*: we can predefine quota value request for new accounts in an AWS organization

  - We can create a CloudWatch Alarm based on a particular service quota
- Legacy method to increase quotas: create a support ticket selecting service quota increase
- We can request service quota increase from the CLI as well. Reference API: https://awscli.amazonaws.com/v2/documentation/api/latest/reference/service-quotas/request-service-quota-increase.html