S3 - Creating a personalised and publicly accessed website using S3 bucket

->> Name of the bucket should be UNIQUE
->> ACLs should be enabled for the general access of the objects like other AWS accounts

> An **ACL** = a set of rules that decides who can get access to a resource.
>
> Enabling ACLs in this S3 setup lets you control who can access and do things with the objects (i.e. website files) you upload into your bucket.
>
> With ACLs, different AWS accounts can own and control different files in your bucket.

->> Uncheck the _Block all public access_

->> Upload the project files and folders to the bucket

->> Configure your S3 bucket for static website **hosting**

- By configuring your S3 bucket for hosting, we're telling this bucket: "please create a URL that will take anyone to a page that displays the HTML file I just uploaded."

Go to bucket -> Properties -> Configure the following settings:

- **Static web hosting: Choose Enable.**
- **Hosting type: Choose Host a static website.**
- **Index document: Enter index.html**

What's a bucket website endpoint?
A bucket website endpoint is just like a regular website URL. It lets people visit your S3 bucket as a website.

->> Visit your public website link.

Upon clicking the link - > we get the 404 error:

403 Forbidden

- Code: AccessDenied
- Message: Access Denied
- RequestId: 71XX0GXVGK09GEHZ
- HostId: STfqzrZJBlWU0PyClvuwDxsrubauxQVx8Fqu0WQH7cy5KBN1aHge5oDB9Z7P2nSZQW9T3rIP1+I=
-

Why did I get this error?
Objects (in this case, the HTML and images files you uploaded) are private by default. This default setting helps keep your account's data secure.

The error message you're seeing is telling you that your static website is being hosted by S3, but the actual HTML/image files you've uploaded are still private. It's kind of like having a bucket on display, so everyone can see the bucket - but the contents are covered up, preventing anyone from seeing what's inside.

To solve this error, we need to set the permission of the objects to public - this is why we enabled **ACLs** in Task 1!

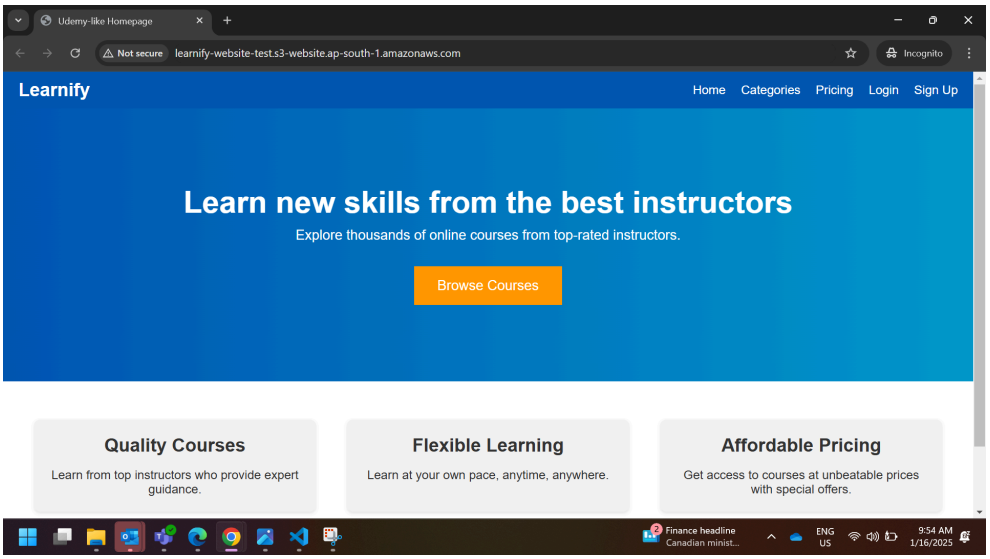
Making the objects public:

->> go to objects and select index.html -> go to Actions -> select ***Make public using ACL***

Similarly do it for the folder as well..

That's it. Now the website is live.

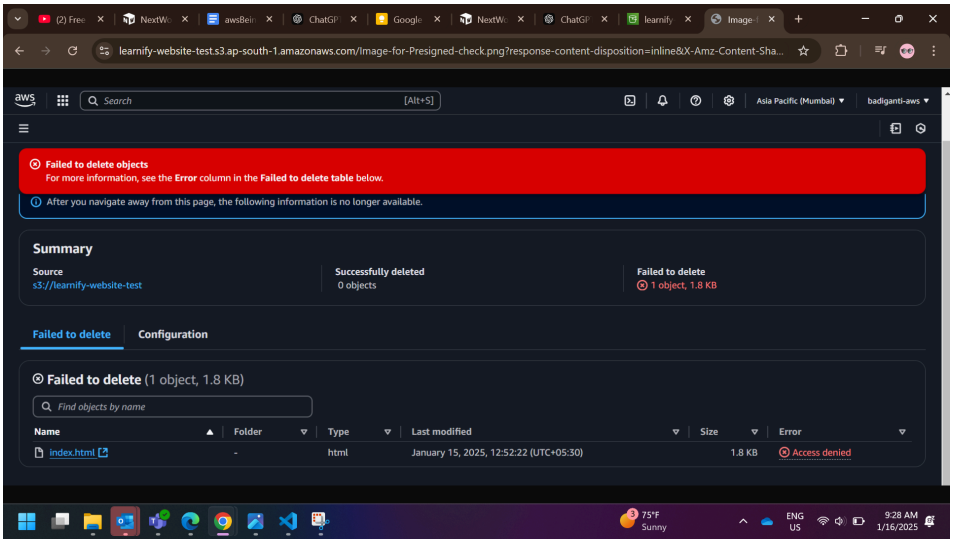
Below is the screenshot of our sample website hosted statically on aws.



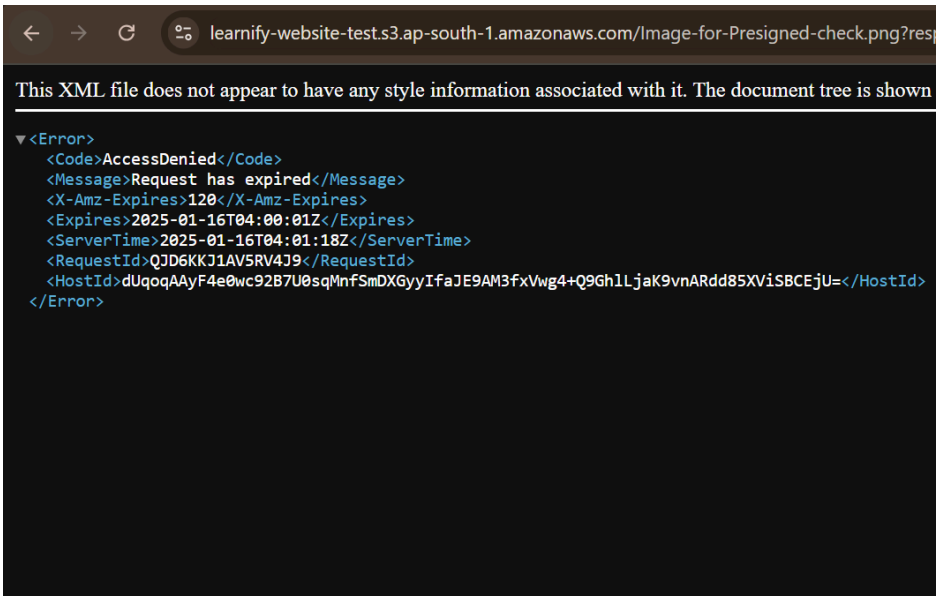
More into S3

Pre-signed URLs :

- **Definition**: A pre-signed URL is a temporary URL generated to grant time-limited access to objects in a private S3 bucket without requiring AWS credentials.
- **Use Cases**:
    - Sharing objects securely with external users.
    - Enabling temporary access to files for downloads or uploads.
- **Features**:
    - Time-limited: You can specify the expiration time for the URL.
    - Permission-limited: Access is restricted to the specified operation (e.g., GET or PUT).



The above image is obtained by using the created Pre-signed URL.



This error occurs after the expiration time set in the pre-signed URL has passed. (Request has expired)

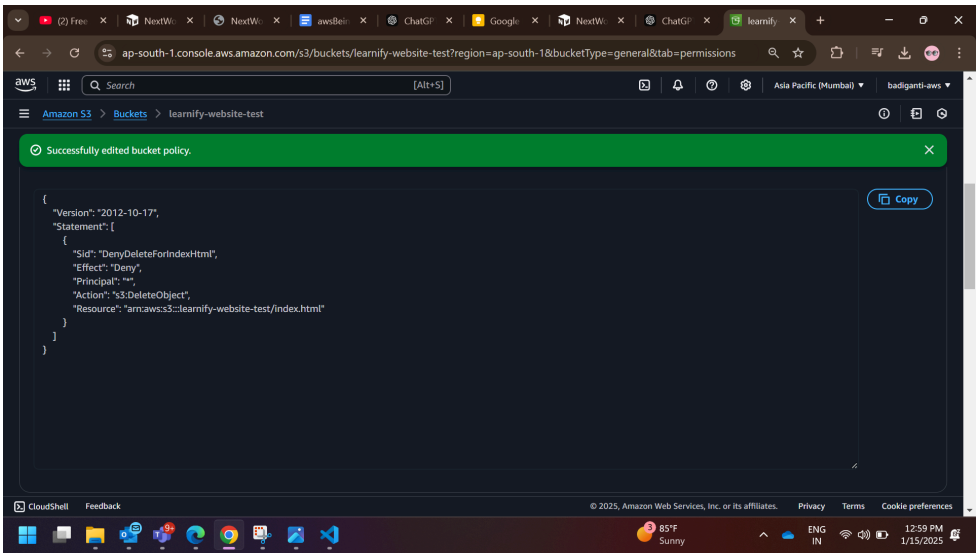
Bucket policy for secure access control :

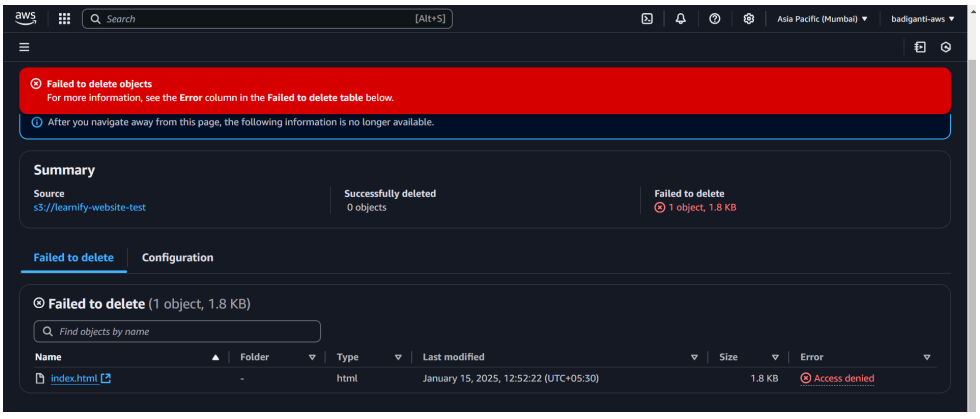
Using the bucket policy to secure the objects.

Example :

Let's say we need to secure our **index.html** file and that we do not want anyone to DELETE the file..

So now we can write the bucket policy to do this.





When tried to delete the index.html

Key Components in Policy making :

1. `Version`: **Policy language version (always `"2012-10-17"`).**
2. `Statement`: **List of rules; can include multiple.**
3. `Sid`: **Optional unique identifier for the statement.**
4. `Effect`: **Determines if the rule `Allow`s or `Deny`s the action.**
5. `Principal`: **Specifies who the policy applies to (`*` for everyone or specific users/roles).**
6. `Action`: **Defines what actions are allowed/denied (e.g., `s3:GetObject`, `s3:DeleteObject`).**
7. `Resource`: **Specifies which bucket or objects the rule applies to.**
8. `Condition`: **Optional constraints (e.g., IP, date, user-agent).**

Updating your website on Amazon S3:

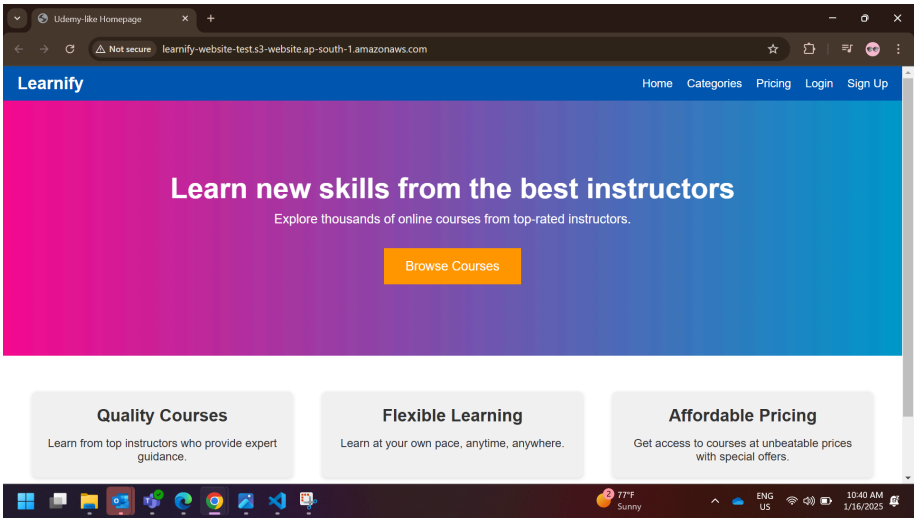
Upload the index.html without deleteing the existing file..
And make the file public using ALCs again.

That's how we can simply update our files. (a simple reupload and making ALCs public)

Bucket Versioning:

Bucket versioning in Amazon S3 is a feature that allows you to keep multiple versions of an object in the same bucket. When versioning is enabled for a bucket, S3 stores every version of an object, including the previous ones, whenever you upload or modify an object. This ensures that you can retrieve older versions of an object if needed.

By default, bv is not enabled. By enabling the bv, changed and deleted versions of the files are saved.



Updated styles.css

Route 53:

Route 53 is a scalable and highly available Domain Name System (DNS) web service offered by Amazon Web Services (AWS). It performs the following functions:

1. **DNS Management**: It translates friendly domain names (like `www.example.com`) into IP addresses (like `192.0.2.1`) that computers use to connect to each other.
2. **Domain Registration**: It allows you to register new domain names and manage your domain settings.
3. **Health Checks and Monitoring**: It can monitor the health of your resources (like web servers) and automatically route traffic to healthy endpoints.
4. **Traffic Routing**: It provides different routing policies for distributing traffic across multiple resources, such as weighted routing, latency-based routing, and geolocation routing.

Certainly! Here's a summarized table that includes key **features** of **Amazon Route 53**, **common record types**, and their descriptions for easy reference in your AWS notes:

Route 53 Summary Table

Feature/Concept	Description
DNS Management	Translates domain names to IP addresses to enable communication between devices on the internet.
Domain Registration	Allows you to register new domain names or transfer existing ones to Route 53.
Health Checks	Monitors the health of your resources (e.g., web servers) and routes traffic away from unhealthy resources.
Routing Policies	Defines how Route 53 handles DNS requests based on different policies: Simple, Weighted, Latency, Geolocation, Failover.
Scalability	Can scale to handle a large volume of DNS queries, suitable for applications of any size.

Integration with AWS	Fully integrates with other AWS services (e.g., EC2, S3, CloudFront, Elastic Load Balancing) for streamlined resource management.
Global Infrastructure	Provides low-latency DNS resolution through a globally distributed network of DNS servers.
Traffic Routing	Route traffic based on user location, load balancing, or health checks to improve performance and availability.
Cost-Effective	Pay-as-you-go pricing model, with charges based on the number of hosted zones, queries, and health checks.

Common Record Types in Route 53

Record Type	Description	Use Case Example
A Record	Maps a domain name to an **IPv4** address (e.g., `192.0.2.1`).	`www.example.com -> 192.0.2.1` (Static website)
AAAA Record	Maps a domain name to an **IPv6** address (e.g., `2001:0db8::1`).	`www.example.com -> 2001:0db8::1` (IPv6 website)
CNAME Record	Maps a domain name to another domain name (e.g., redirecting traffic to another domain). **Does not point to an IP address.**	`www.example.com -> example.com` (Alias record)
TXT Record	Used to store arbitrary text data, often for verification purposes (e.g., SPF, DKIM).	`example.com -> "v=spf1 include:_spf.google.com ~all"`

MX Record	Specifies mail servers that handle email for the domain.	`example.com -> mail.example.com` (Email routing)
NS Record	Specifies authoritative DNS servers for a domain. Typically provided by your DNS hosting provider.	`example.com -> ns-123.awsdns.com`
SOA Record	Specifies the **Start of Authority** for the domain, including information about the DNS zone, such as the primary DNS server and the time-to-live (TTL).	Auto-generated in hosted zones
PTR Record	Used for **reverse DNS lookups**, mapping an IP address to a domain name.	`192.0.2.1 -> example.com` (Reverse DNS lookup)
SRV Record	Specifies services offered by a domain, including the protocol, port, and target machine for the service.	`_sip._tcp.example.com` (Service discovery)
CAA Record	Specifies which Certificate Authorities (CAs) are allowed to issue SSL/TLS certificates for the domain.	`example.com -> 0 issue "letsencrypt.org"` (Security)
Alias Record	Similar to a CNAME record, but allows mapping to AWS resources (e.g., ELB, CloudFront, S3 buckets). Can be used at the root domain (which CNAME cannot).	`example.com -> s3-website-us-east-1.amazonaws.com`

Routing Policies

Routing Policy	Description	Use Case Example
Simple Routing	Directs traffic to a single resource.	Pointing `www.example.com` to one EC2 instance.
Weighted Routing	Distributes traffic across multiple resources based on assigned weights.	80% traffic to primary EC2, 20% to a secondary instance.
Latency-based Routing	Routes traffic to the resource with the lowest latency based on the user's location.	Direct traffic to EC2 in closest AWS region.
Geolocation Routing	Routes traffic based on the geographical location of the user.	Route U.S. traffic to U.S. servers, EU traffic to EU.
Failover Routing	Routes traffic to a secondary resource if the primary resource becomes unhealthy.	Primary web server goes down, traffic automatically rerouted to backup server.

See you on my next one! 👋

21-01-25

IAM in aws : A service that enables you to securely manage access to AWS resources for users, groups, and services.

- **Core Features**:
    - Centralized control over AWS account permissions.
    - Secure access to AWS services and resources.
    - Supports multi-factor authentication (MFA).

Key Concepts

1. Users

- Represents individual people or applications.
- Each user has a unique identity and can have login credentials (passwords or access keys).

2. Groups

- A collection of IAM users.
- Simplifies permission management by assigning policies to groups instead of individual users.

3. Roles

- Temporary access permissions for AWS services or external users.
- Ideal for granting permissions to services like EC2 or Lambda without sharing credentials.

4. Policies

- JSON documents that define permissions.
- Types of policies:
    - **Managed Policies**: Created and managed by AWS or users.
    - **Inline Policies**: Embedded directly into a user, group, or role.

5. Access Keys

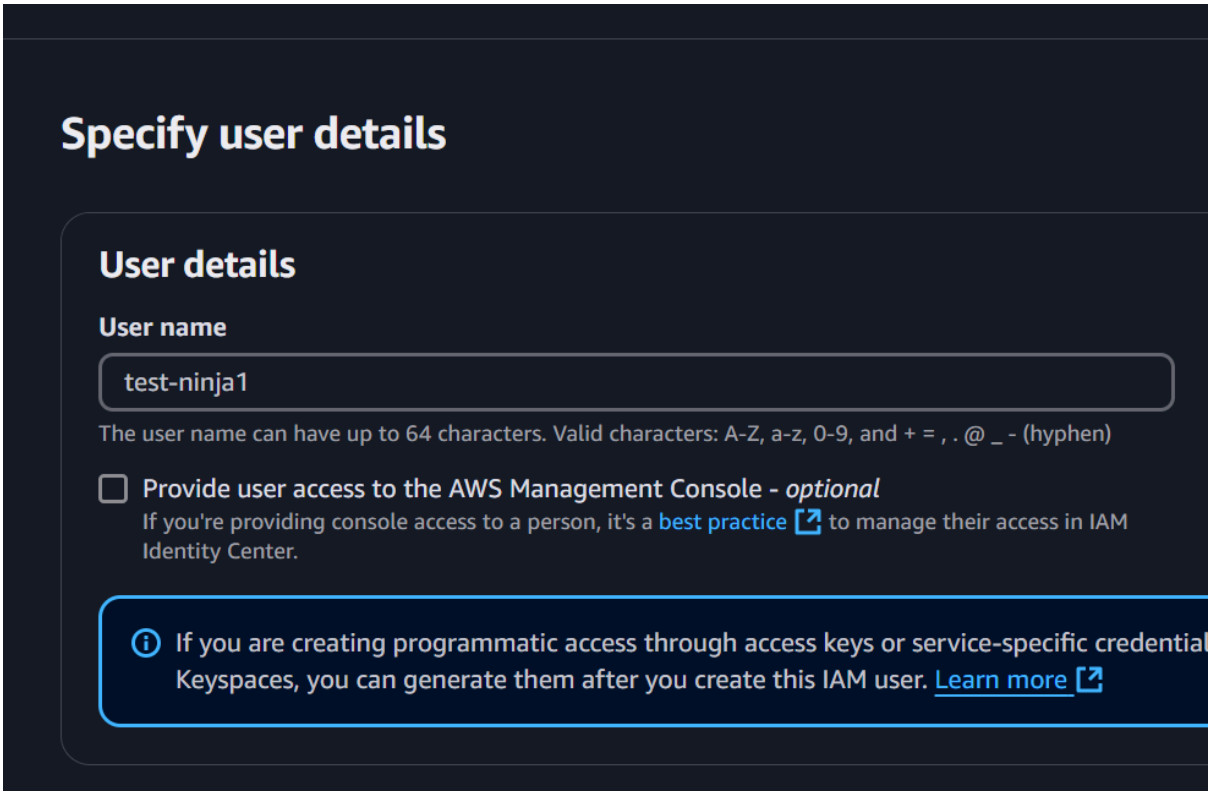
- Provide programmatic access to AWS services.
- Consist of an Access Key ID and a Secret Access Key.

6. MFA (Multi-Factor Authentication)

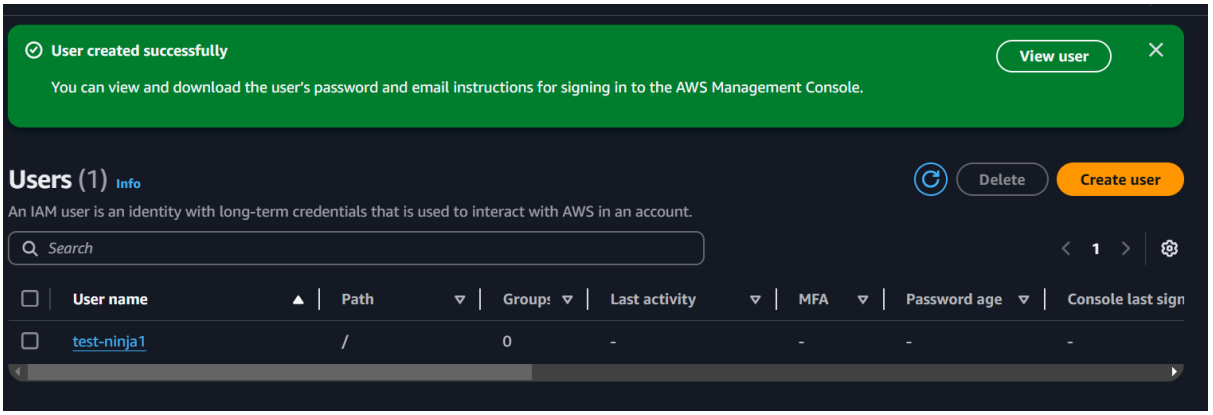
- Adds an extra layer of security by requiring a second authentication factor.
- Types: Virtual MFA devices, hardware MFA devices, and SMS MFA.

Initially I don't have any IAM users or groups, I'm currently logging in through root user only.

First, I'm gonna create a test-user for IAM.



Permissions options - > Add user to group



test-user created.

(

When creating a user, **ARN** will be generated .

ARN : An **Amazon Resource Name (ARN)** is a unique identifier used in AWS to specify resources within the AWS ecosystem. ARNs are used across AWS services to grant permissions, specify resources, and manage configurations.

ARN examples :

IAM User : arn:aws:iam::123456789012:user/JohnDoe

S3 bucket : arn:aws:s3:::my-example-bucket

Ec2 instance : arn:aws:ec2:us-east-1:123456789012:instance/i-0abcd1234ef567890

Why ARNs Are Important

1. **Resource Identification**:
    - ARNs uniquely identify AWS resources, making them critical for referencing resources in policies or APIs.
2. **Access Control**:
    - ARNs are used in **IAM policies** to specify which resources users or services can access.

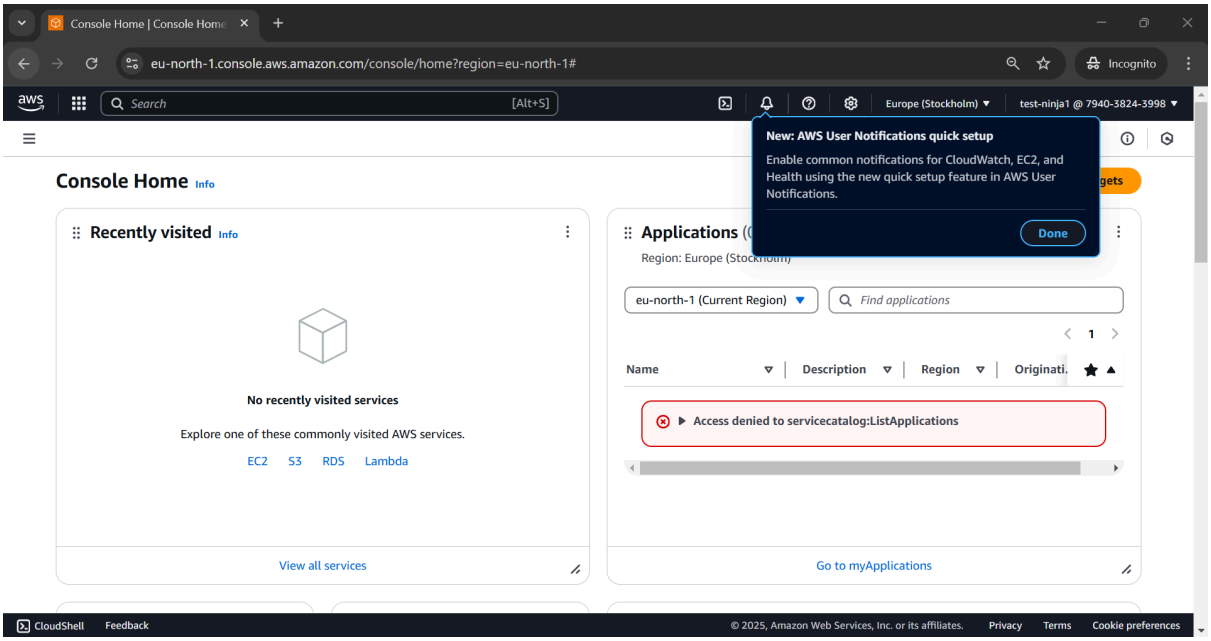
)

View the user - > security credentials -> ***Enable console access***

then, we'll get the following for that test user :

Console sign-in URL, password (auto generated/set)

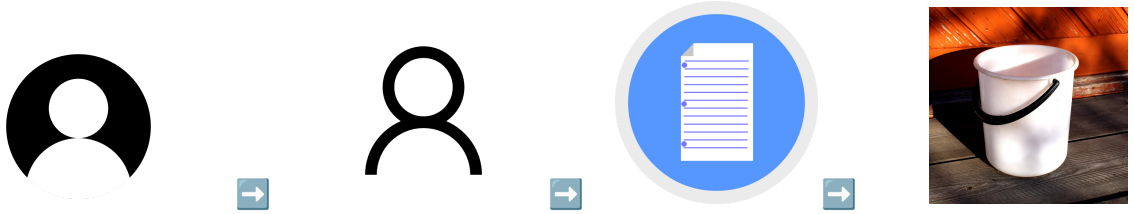
Copy the url and sign in..



Initially, this test user will not have any access to any type of application or the resource.

So, for any user, group created we need to create a particular POLICY to access a specific resource like s3, ec2 etc ..

Creating a policy for test-ninja1 to have full access to S3 [listing, viewing, creating, deleting etc..]



Root acc → test-ninja1 → IAM policy → access to S3

Creating the policy :

Go to IAM - > Policies -> Create policy - > json (better to edit)

Edit the json

```
{
        "Version": "2012-10-17",
        "Statement": [
                {
                        "Sid": "AllowFullS3Access",
                        "Effect": "Allow",
                        "Action": "s3:*",
                        "Resource": "*"
                }
        ]
}
```

Version - standard AWS syntax we need to follow [Specifies the version of the policy language. `"2012-10-17"` is the latest and most commonly used version.]

Sid - can be name of what that policy is for

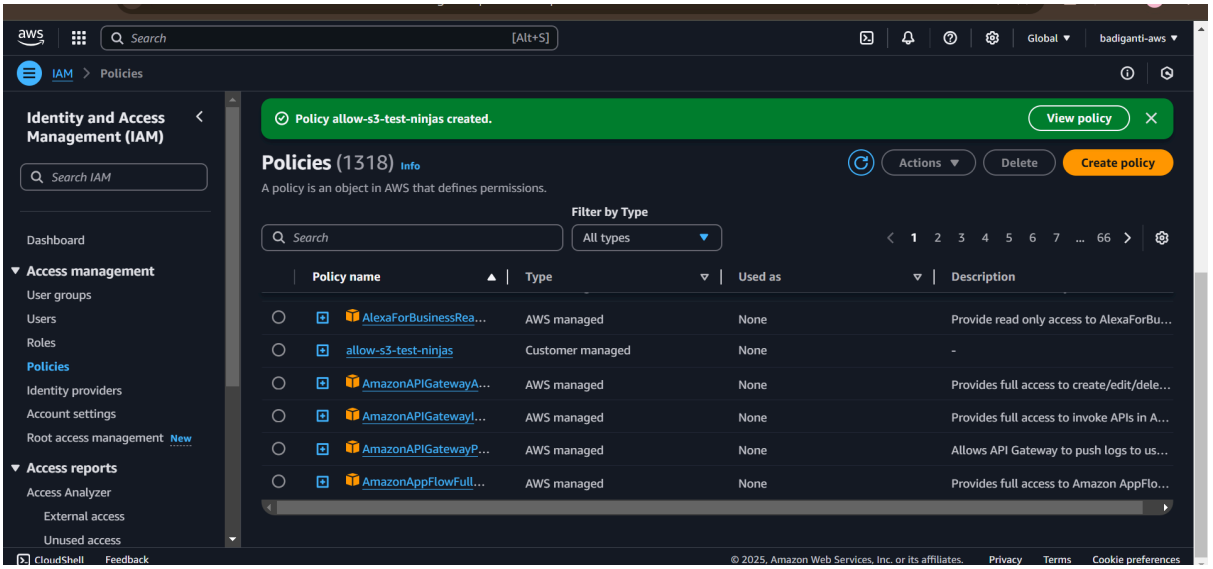
Action : allow or deny the access

specifies the S3 actions that are allowed or denied. `"s3:*"` means all possible actions on S3, such as:

- `s3:ListBucket` (list objects in a bucket),
- `s3:GetObject` (retrieve objects),
- `s3:PutObject` (upload objects), and more.

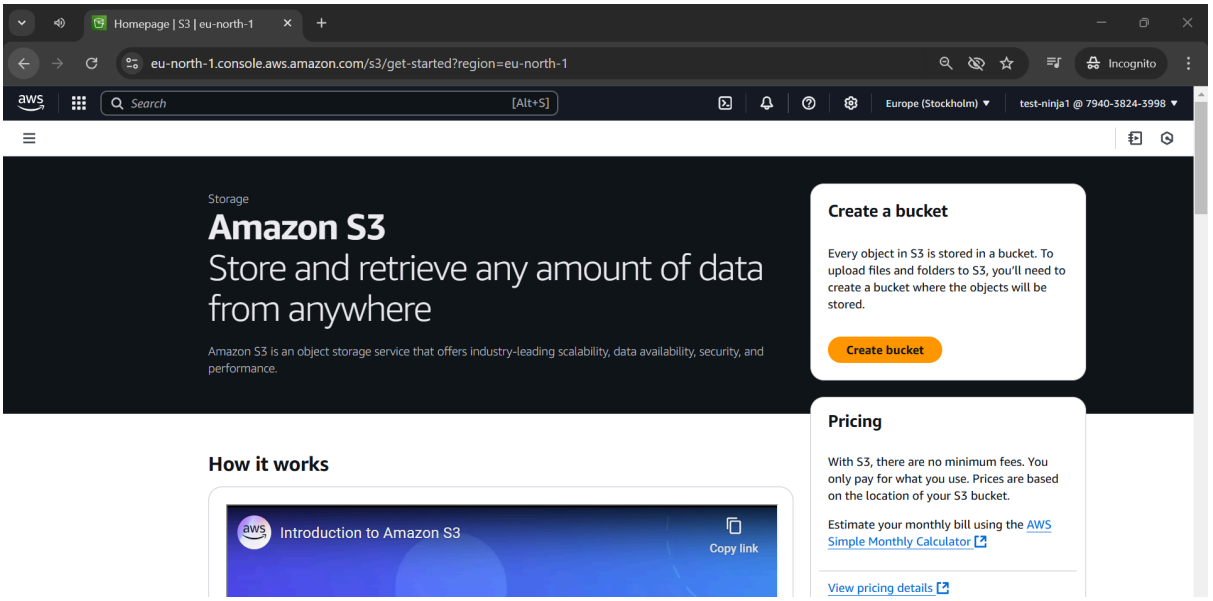
Resource : Specifies the AWS resources the policy applies to. `"*"` means all resources, i.e., all S3 buckets and objects in the account.

-> Next -> Give a name to the policy -> Create policy



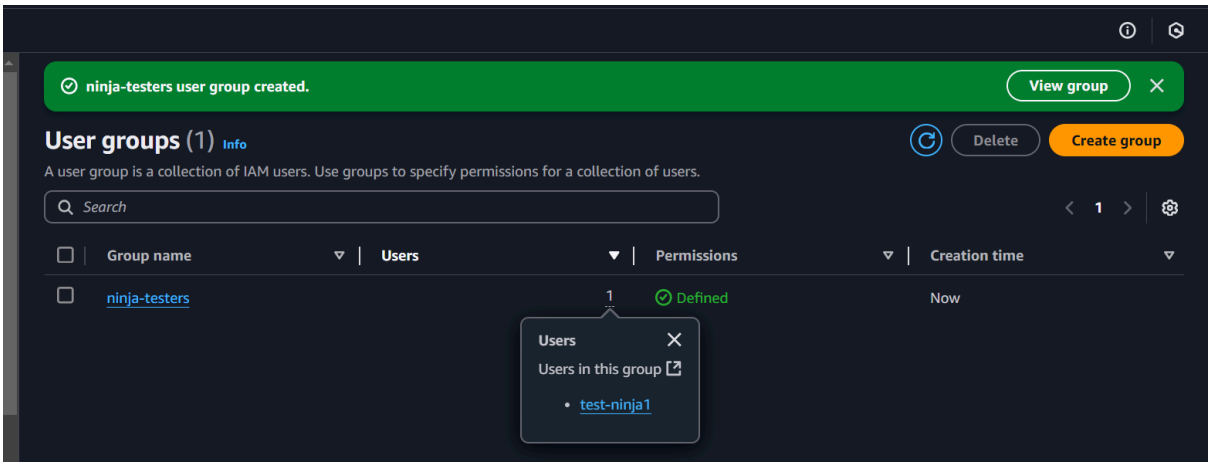
Policy created.

Log in to the test-ninja1 and check for s3 access ..



Creating a group and adding users to it :

Go to IAM -> User groups -> NAme the group -> add users to it -> attach the policies if you want



AWS Organizations : AWS Organizations is a service that enables centralized management of multiple AWS accounts, simplifying governance, security, and cost management.

Key Features:

1. **Centralized Management**:
   Manage multiple AWS accounts from one location.

2. **Service Control Policies (SCPs)**:
   - Enforce permissions boundaries at the account or Organizational Unit (OU) level.
   - Restrict or allow specific AWS services/actions.
3. **Consolidated Billing**:
   - Combine usage across accounts for volume discounts.
   - Receive a single bill for all accounts.
4. **Organizational Units (OUs)**:
   - Group accounts into OUs for easier policy application.
   - Hierarchically structure accounts (e.g., by team, environment).
5. **Delegated Administration**:
   Assign account-specific management roles for particular services without full access to the organization.

6. **Integration with AWS Services**:
   Works with AWS Control Tower, AWS Config, AWS Service Catalog, etc., for governance and compliance.

Basically in an organization there will be different teams like dev, test, prod etc..
So for every team the AWS ORG will create Organization Units (OU) separately to have different environments..

And for every OU a separate ROOT account is required.
So at first lets say we have one ROOT account.

Then we want to have two teams like dev-OU and test-OU.

Now with the actual ROOT account we can create the dev-OU and for the test-OU we will have to create a new account being in the ROOT account.

(give a new email id, like we do in creating a ROOT account at the start, now copy the email after the account creation and use forgot-password to log in and set a new pw.. And that's how you'll be logged into the new ROOT account for test-OU)

DEtailed steps :

Steps to Create Organizational Units (OUs) for Dev and Test Teams

1. ***Start with the Primary ROOT Account**:*
    - *Begin by logging into the AWS Management Console using the **primary root account** (the one created when setting up the AWS Organization).*
    - *This account serves as the master account for managing the organization and creating OUs.*
2. ***Understand the Team Requirements**:*
    - *Identify the teams or environments you need to create (e.g., `dev`, `test`, `prod`, etc.).*
    - *Plan to create separate **Organizational Units (OUs)** for each team or environment.*
3. ***Create the First OU (e.g., Dev-OU)**:*
    - *In the AWS Organizations console:*
        1. *Navigate to the **Organizations** section.*
        2. *Click on **Organizational units** and create a new OU named `dev-OU`.*
    - *No additional accounts need to be created for this step if it will use existing resources/accounts.*
4. ***Create a New Account for Another OU (e.g., Test-OU)**:*
    - *For the `test-OU`, a new AWS account is required. To create the account:*
        1. *Go to the **Accounts** section in the AWS Organizations console.*
        2. *Click **Add an AWS Account** and select **Create an AWS Account**.*
        3. *Provide the following details:*
            - ***Account Name**: `test-OU`*
            - ***Email Address**: A new, unique email address that has not been used with AWS before (e.g., `test-OU@example.com`).*
        4. *Confirm and submit the details.*
    - *Once the account is created, it will automatically be added to your organization.*

5. ***Set Up the New ROOT Account for Test-OU**:*

    - *Use the email address provided during the account creation to log in:*
        1. *Go to the AWS login page.*
        2. *Use the "Forgot Password" option to set a new password for the new root account.*
        3. *Log in to the `test-OU` root account with the updated credentials.*
6. ***Organize the Accounts into OUs**:*

    - *Move the new account created for `test-OU` into the respective `test-OU` Organizational Unit in the AWS Organizations console:*
        1. *Navigate to the **Accounts** section.*
        2. *Select the account and move it to the desired OU.*
7. ***Repeat for Additional OUs**:*

    - *Follow the same process for creating other OUs (e.g., `prod-OU`), including creating new accounts if needed.*

Key Notes:

- ***Email Addresses**: Each AWS account must have a unique email address. Plan and use a naming convention for email addresses to keep them organized.*
- ***Root Account Management**: Always secure the root account credentials with strong passwords and multi-factor authentication (MFA).*
- ***Service Control Policies (SCPs)**: After creating OUs, apply SCPs to enforce permissions and restrictions for each OU as required.*

EC2 : Amazon EC2 is a web service provided by AWS that offers resizable compute capacity in the cloud. It is designed to simplify cloud computing by providing scalable resources with minimal friction.

Key features :

Elasticity and Scalability

- *Quickly scale up or down based on demand.*
- *Offers auto-scaling capabilities for dynamic adjustments.*

Customizable

- *Choose OS, storage, and networking configurations.*

Pay-as-you-go

- *Pricing is based on usage, with options like on-demand, reserved, and spot instances.*

Secure

- *Integrated with AWS Identity and Access Management (IAM).*

Components of EC2:

Component	***Description***
Instances	*Virtual servers running applications; configurable based on CPU, memory, storage, and networking needs.*
Amazon Machine Image (AMI)	*Pre-configured templates for creating instances, including OS and applications. Types: Public, Private, Marketplace.*
Instance Types	*Categorized for workloads: General Purpose, Compute Optimized, Memory Optimized, Storage Optimized, Accelerated Computing.*
Key Pair	*Used for secure login; consists of a Private Key (user-stored) and a Public Key (stored in AWS).*
Elastic Block Store (EBS)	*Persistent block storage. Volume types: General Purpose SSD (gp3, gp2), Provisioned IOPS SSD (io1, io2), Throughput Optimized HDD (st1), Cold HDD (sc1).*
Security Groups	*Virtual firewall controlling traffic; rules include protocols (TCP, UDP, ICMP), port ranges, and IP addresses.*

Elastic IP	*Static IPv4 address associated with an account; can be reassigned between instances.*
Placement Groups	*Logical grouping of instances for workload optimization: Cluster (low-latency), Spread (fault tolerance), Partition (isolated partitions).*

Launching EC2 instance :

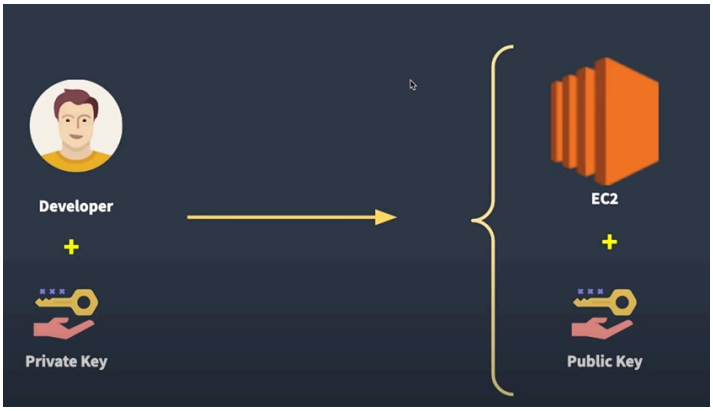
Go to EC2 - > Give name -> Select your AMI -> aRchitecture -> Instance type -> Key pair

Key pair : You can use a key pair to securely connect to your instance. Ensure that you have access to the selected key pair before you launch the instance.

Create a new key pair, -> Name the key-pair -> select RSA -> .pem -> Click on the create key pair.

Automatically a **.pem** file will be downloaded.

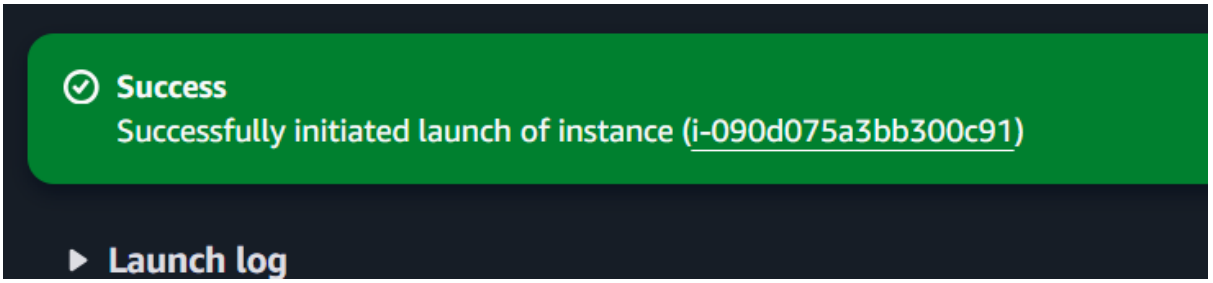
This key pair is used to ssh from remote,



Public key will be with the Ec2 instance, and the downloaded .pem is the private key which will be with the us.

- > Keep the network settings to default.
- > Allow ssh traffic from anywhere

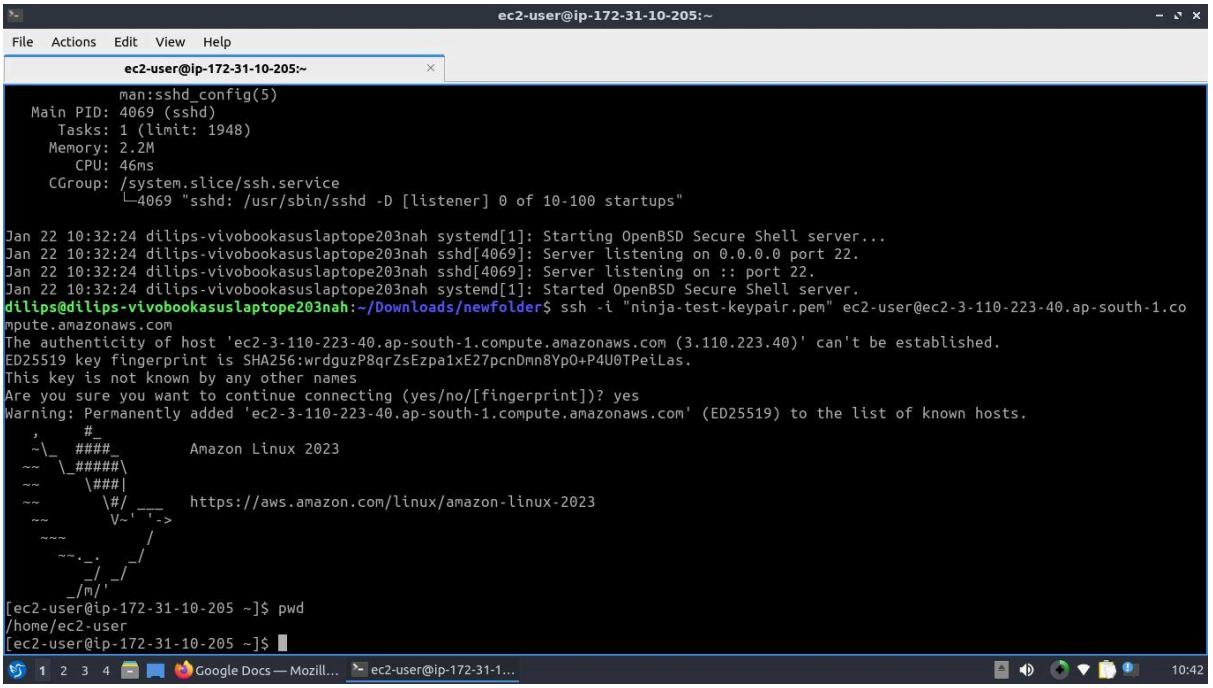
-> Configure the storage as needed. -> Launch instance



Instance id

Connecting to the instance via SSH:

- ➔ Open the running instance..
- ➔ Click on connect
- ➔ Go to SSH client
- ➔ Now open the ssh client in the computer/remote
- ➔ ensure your key is not publicly viewable -> **`chmod 400 "ninja-test-keypair.pem"`**
- ➔ Make sure the ssh is enabled in the remote host.
- ➔ Check the status of your ssh (linux version - systemctl status ssh)
- ➔ Once it is active, you can ssh using the public dns provided by the instance. Ex: ssh -i "ninja-test-keypair.pem" ec2-user@ec2-3-110-223-40.ap-south-1.compute.amazonaws.com



Finally, terminate your instances after use.

See you on my next one! 👋