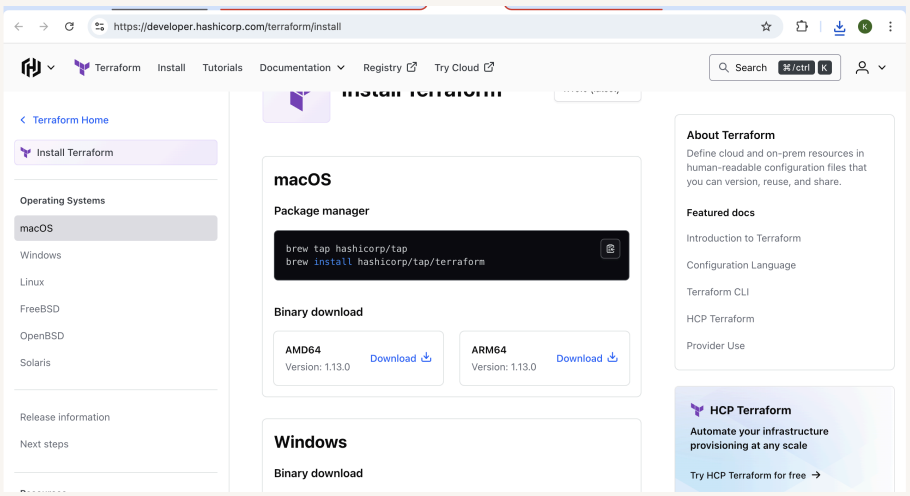
nextwork.org

Create S3 Buckets with Terraform

Kehinde Abiuwa



Introducing Today's Project!

In this project, I will demonstrate how to install and configure Terraform, set up AWS credentials in the terminal, and use Terraform to create and manage S3 buckets. I will also show how to upload files to S3 with Terraform. The goal is to gain hands-on experience with Terraform for AWS resource management and automation.

Tools and concepts

Services I used were: AWS S3 (bucket creation, static website hosting, object storage). AWS IAM (user and access keys for auth). AWS STS (GetCallerIdentity during credential validation). AWS CLI v2 (to configure creds and verify results). Terraform with the AWS provider. Key concepts I learnt include Infrastructure as Code (IaC) and how Terraform represents it with: Provider, resource, and data blocks (e.g., aws_s3_bucket, aws_s3_object, website config). State management (terraform.tfstate) and provider locking (.terraform.lock.hcl). The safe workflow: terraform init → plan → apply (prepare, preview, perform). Idempotency & change management (edit config, re-apply to update real infrastructure). S3 public access controls (Public Access Block vs. bucket policies) and implications for website hosting. Credential management (profiles, env vars, and fixing InvalidClientTokenId errors). Content deployment via code using aws_s3_object to upload files as part of the plan.

Project reflection

This project took me approximately 50 minutes. The most challenging part was debugging the Terraform errors—first the “timeout while waiting for plugin to start” (fixed by re-initializing the provider and clearing macOS quarantine/caches) and then the InvalidClientTokenId AWS auth issue (solved by installing AWS CLI, configuring a profile with valid access keys, and verifying with aws sts get-caller-identity).

It was most rewarding to run terraform apply and see the S3 bucket and website config created, then upload image.png via aws_s3_object and download it from the S3 console to confirm the deployment worked end-to-end.

I chose to do this project today because... Something that would make learning with NextWork even better is...

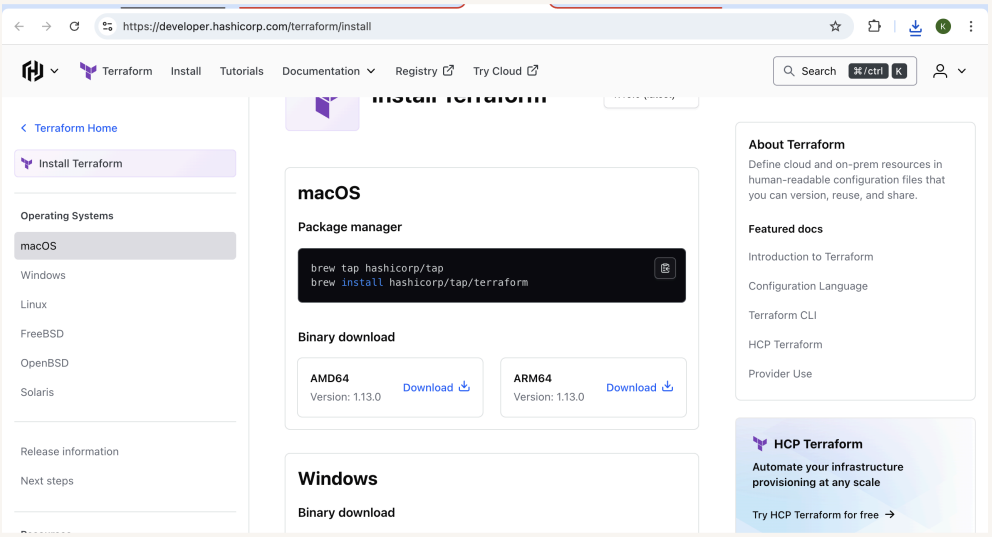
K
Kehinde Abiuwa
NextWork Student

Introducing Terraform

Terraform is an open-source Infrastructure as Code (IaC) tool that lets you define and manage cloud resources (like servers, databases, and storage) using simple configuration files. Instead of manually setting things up in the cloud, you write instructions in Terraform, and it automatically creates, updates, or deletes the resources for you. It works with many providers (like AWS, Azure, Google Cloud), making it easier to automate and manage infrastructure consistently and reliably.

Terraform is one of the most popular tools used for Infrastructure as Code (IaC), which is the practice of managing and provisioning infrastructure through code instead of manual processes. With IaC, you write configuration files that describe the resources you need (like servers, networks, or storage), and tools like Terraform automatically create and manage those resources for you.

Terraform uses configuration files to describe the infrastructure you want to create, change, or manage. These files are written in HashiCorp Configuration Language (HCL) and act as instructions for Terraform. main.tf is the primary configuration file where you define the resources (like AWS S3 buckets, EC2 instances, or networking components) that Terraform will build. It usually serves as the starting point of your Terraform project.

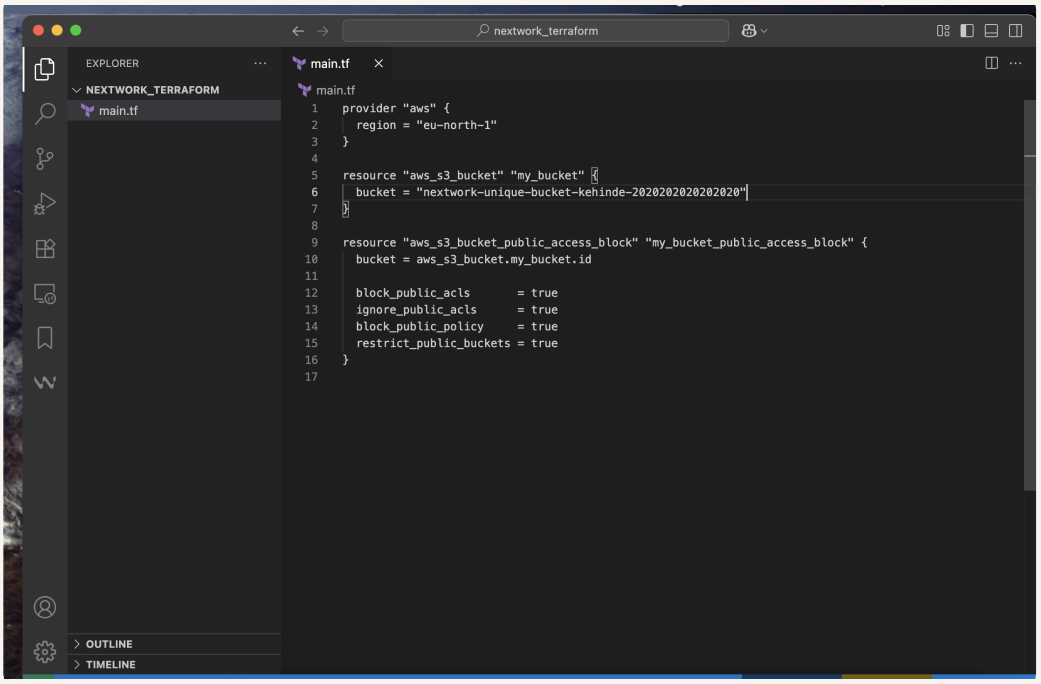

https://developer.hashicorp.com/terraform/install
Terraform
Install
Tutorials
Documentation
Registry
Try Cloud
Search
⌘/ctrl K
Terraform Home
Install Terraform
Operating Systems
macOS
Windows
Linux
FreeBSD
OpenBSD
Solaris
Release information
Next steps
macOS
Package manager
brew tap hashicorp/tap
brew install hashicorp/tap/terraform
Binary download
AMD64
Version: 1.13.0
Download
ARM64
Version: 1.13.0
Download
Windows
Binary download
About Terraform
Define cloud and on-prem resources in human-readable configuration files that you can version, reuse, and share.
Featured docs
Introduction to Terraform
Configuration Language
Terraform CLI
HCP Terraform
Provider Use
HCP Terraform
Automate your infrastructure provisioning at any scale
Try HCP Terraform for free

Configuration files

The configuration is structured in blocks. Each block tells Terraform what to do: A provider block defines which cloud provider (AWS) and region to use. A resource block defines what you want to create (like an S3 bucket). Additional resource blocks can manage settings for those resources (like blocking public access to the bucket). The advantage of doing this is that the setup is clear, reusable, and easy to change. Instead of manually creating resources in AWS, you can just edit the configuration file, and Terraform will automatically update your infrastructure to match.

My main.tf configuration has three blocks

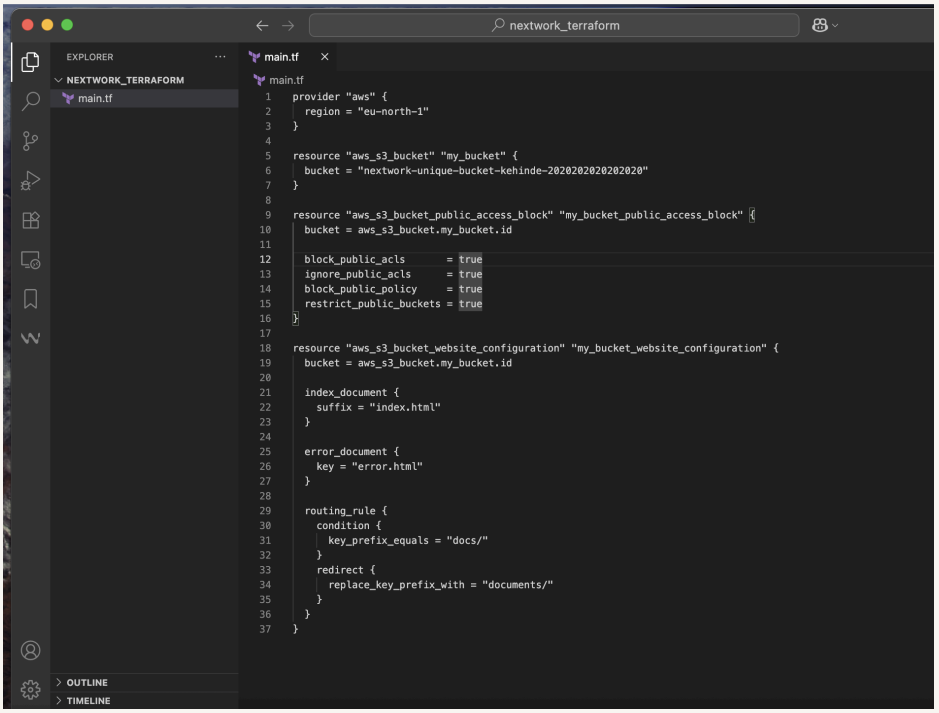
The first block indicates the AWS provider and sets the region to eu-north-1, telling Terraform where to create the resources. The second block provisions an Amazon S3 bucket with a unique bucket name. The third block manages the S3 bucket's access settings by blocking all forms of public access, ensuring the bucket stays private and secure.


nextwork_terraform
EXPLORER
NEXTWORK_TERRAFORM
main.tf
main.tf
main.tf
1 provider "aws" {
2 region = "eu-north-1"
3 }
4
5 resource "aws_s3_bucket" "my_bucket" {
6 bucket = "nextwork-unique-bucket-kehinde-2020202020202020"
7 }
8
9 resource "aws_s3_bucket_public_access_block" "my_bucket_public_access_block" {
10 bucket = aws_s3_bucket.my_bucket.id
11
12 block_public_acls = true
13 ignore_public_acls = true
14 block_public_policy = true
15 restrict_public_buckets = true
16 }
17
OUTLINE
TIMELINE

Customizing my S3 Bucket

For my project extension, I visited the official Terraform documentation to explore the aws_s3_bucket resource. The documentation shows that Terraform provides a dedicated aws_s3_bucket resource to define and manage Amazon S3 buckets declaratively. It includes details on available arguments (such as bucket, acl, versioning, etc.), supported attributes, how to configure additional features like lifecycle rules, server-side encryption, and notes regarding any deprecated options or refactoring updates in newer provider versions

I chose to customise my bucket by enabling S3 static website hosting with an index document (index.html), an error document (error.html), and a routing rule that redirects any path starting with docs/ to documents/, because I want to serve a simple static site, show a friendly error page, and keep old links working after a folder rename. When I launch my bucket, I can verify my customization by uploading index.html and error.html and then visiting the bucket's website endpoint (S3 console → Properties → Static website hosting). I should see: the home page loads at the root, a missing page shows my custom error page, a URL like /docs/example.html redirects to /documents/example.html.

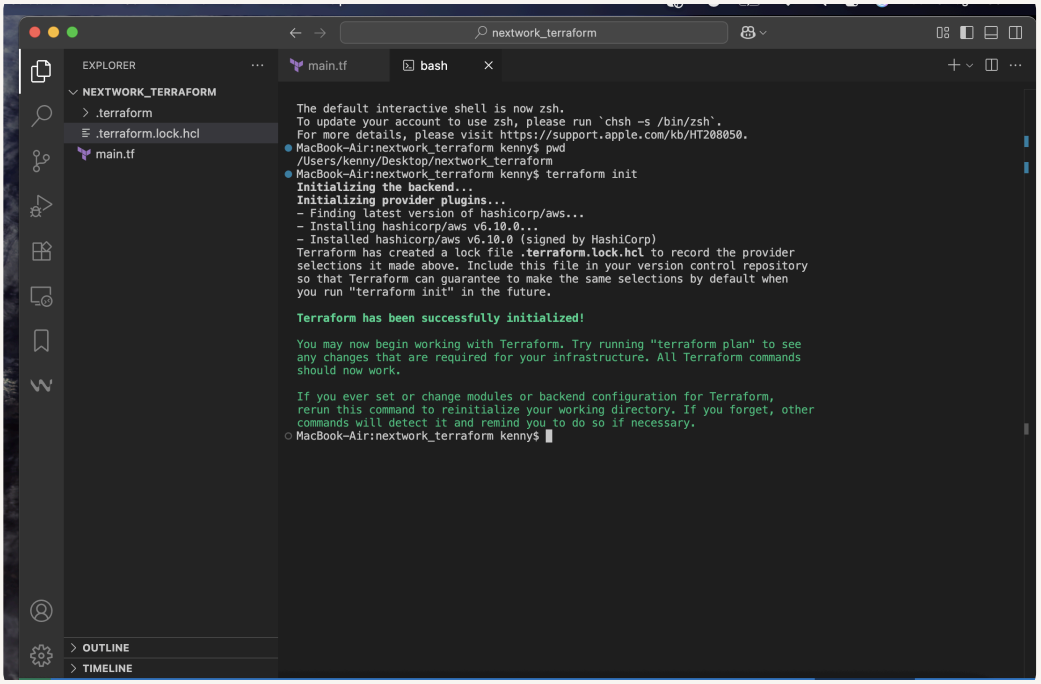

nextwork_terraform
EXPLORER
NEXTWORK_TERRAFORM
main.tf
OUTLINE
TIMELINE
main.tf
provider "aws" {
region = "eu-north-1"
}
resource "aws_s3_bucket" "my_bucket" {
bucket = "nextwork-unique-bucket-kehinde-2020202020202020"
}
resource "aws_s3_bucket_public_access_block" "my_bucket_public_access_block" {
bucket = aws_s3_bucket.my_bucket.id
block_public_acls = true
ignore_public_acls = true
block_public_policy = true
restrict_public_buckets = true
}
resource "aws_s3_bucket_website_configuration" "my_bucket_website_configuration" {
bucket = aws_s3_bucket.my_bucket.id
index_document {
suffix = "index.html"
}
error_document {
key = "error.html"
}
routing_rule {
condition {
key_prefix_equals = "docs/"
}
redirect {
replace_key_prefix_with = "documents/"
}
}
}

Kehinde Abiuwa
NextWork Student

Terraform commands

I ran terraform init to initialize my Terraform project—it sets up the working directory, downloads the required providers (AWS), initializes any modules and the backend (or local state if none is set), and creates the .terraform/ folder and .terraform.lock.hcl for consistent provider versions—so the project is ready to run terraform plan and terraform apply.

Next, I ran terraform plan to safely preview the changes Terraform would make—creating the S3 bucket, applying the website configuration, and setting the Public Access Block



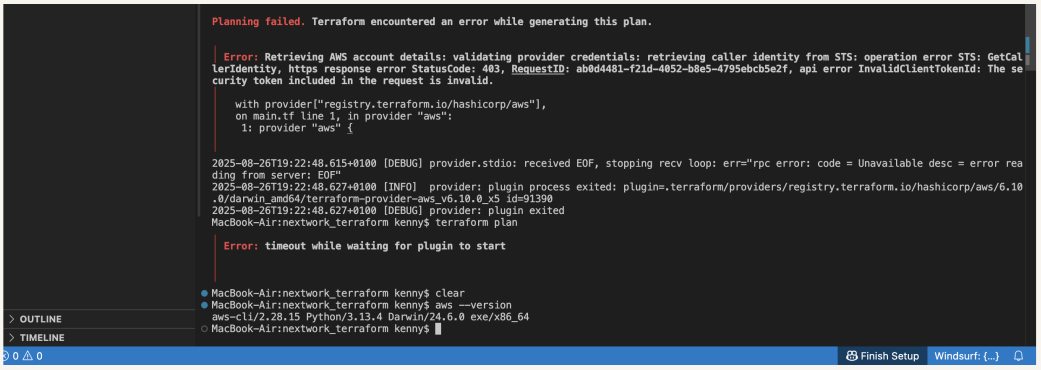
K
Kehinde Abiuwa
NextWork Student

AWS CLI and Access Keys

When I tried to plan my Terraform configuration, I received an error message that says “InvalidClientTokenId: The security token included in the request is invalid” because Terraform called AWS STS to identify my account, but my AWS credentials/profile were incorrect or expired (e.g., wrong access keys or a stale AWS_SESSION_TOKEN), so authentication failed.

To resolve my error, first I installed AWS CLI, which is Amazon’s command-line tool for authenticating to AWS and managing services. It provides the aws command and stores credentials in ~/.aws/credentials and ~/.aws/config—the same locations Terraform reads—so it’s the easiest way to set up valid creds.

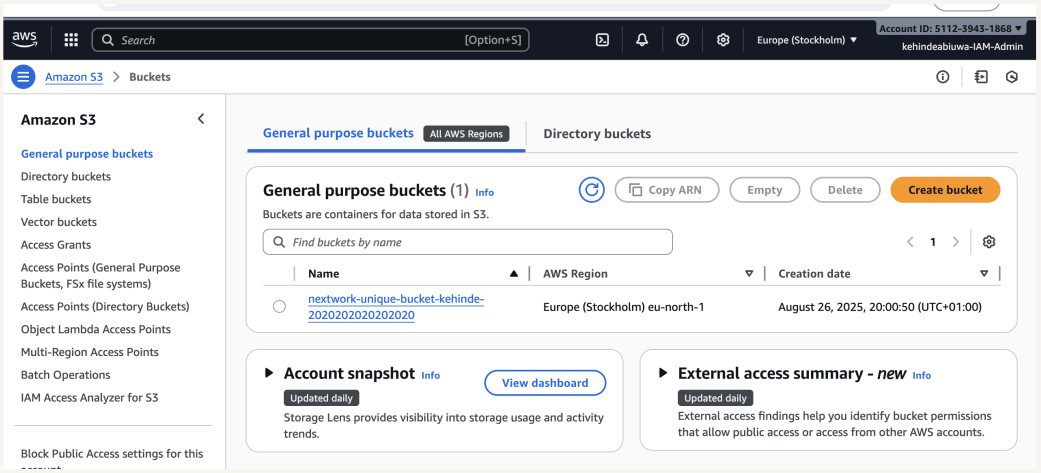
I set up AWS access keys to securely authenticate the AWS CLI and Terraform to my AWS account, so they can sign API requests and create/manage resources during plan and apply without using the web console.


Planning failed. Terraform encountered an error while generating this plan.
Error: Retrieving AWS account details: validating provider credentials: retrieving caller identity from STS: operation error STS: GetCallerIdentity, https response error StatusCode: 403, RequestID: ab0d4481-f21d-4052-b8e5-4795ebcb5e2f, api error InvalidClientTokenId: The security token included in the request is invalid.
with provider["registry.terraform.io/hashicorp/aws"],
on main.tf line 1, in provider "aws":
1: provider "aws" {
2025-08-26T19:22:48.615+0100 [DEBUG] provider.stdio: received EOF, stopping recv loop: err="rpc error: code = Unavailable desc = error reading from server: EOF"
2025-08-26T19:22:48.627+0100 [INFO] provider: plugin process exited: plugin=.terraform/providers/registry.terraform.io/hashicorp/aws/6.10.0/darwin_amd64/terraform-provider-aws_v6.10.0_x5 id=91390
2025-08-26T19:22:48.627+0100 [DEBUG] provider: plugin exited
MacBook-Air:nextwork_terraform kenny$ terraform plan
Error: timeout while waiting for plugin to start
MacBook-Air:nextwork_terraform kenny$ clear
MacBook-Air:nextwork_terraform kenny$ aws --version
aws-cli/2.28.15 Python/3.13.4 Darwin/24.6.0 exe/x86_64
MacBook-Air:nextwork_terraform kenny$
OUTLINE
TIMELINE
0 ⚠ 0
Finish Setup
Windsurf: {...}

Lanching the S3 Bucket

I ran terraform apply to execute the plan and create the real AWS resources defined in main.tf—the S3 bucket, its Public Access Block, and the website configuration—after previewing the changes with terraform plan. Running terraform apply will affect my AWS account by: Creating/modifying/deleting resources to match my config (in this case: creating the S3 bucket and its settings). Updating the Terraform state file (terraform.tfstate) with the real IDs/ARNs so future runs know what exists. Incurring charges for any billable resources and data usage (S3 storage, requests, and data transfer). Applying access controls I defined (e.g., blocking public access or enabling website hosting), which changes who can access the bucket. Recording a change history in AWS (CloudTrail shows API calls made by Terraform's AWS provider).

The sequence is crucial because each command prepares the next one and protects you from accidental changes. terraform init → setup. Downloads/locks the provider(s), configures the backend/state, and makes the working directory usable. Run it first, and again whenever you add/upgrade providers or change the backend. terraform plan → preview. Reads your .tf files and current state, compares to real AWS, and shows the exact actions (create/change/destroy) without changing anything. Use it to verify intent. Optionally save it: terraform plan -out=tfplan. terraform apply → execute. Performs the actions to match your config and updates terraform.tfstate. Best practice: apply the saved plan (terraform apply tfplan) so you execute exactly what you reviewed.

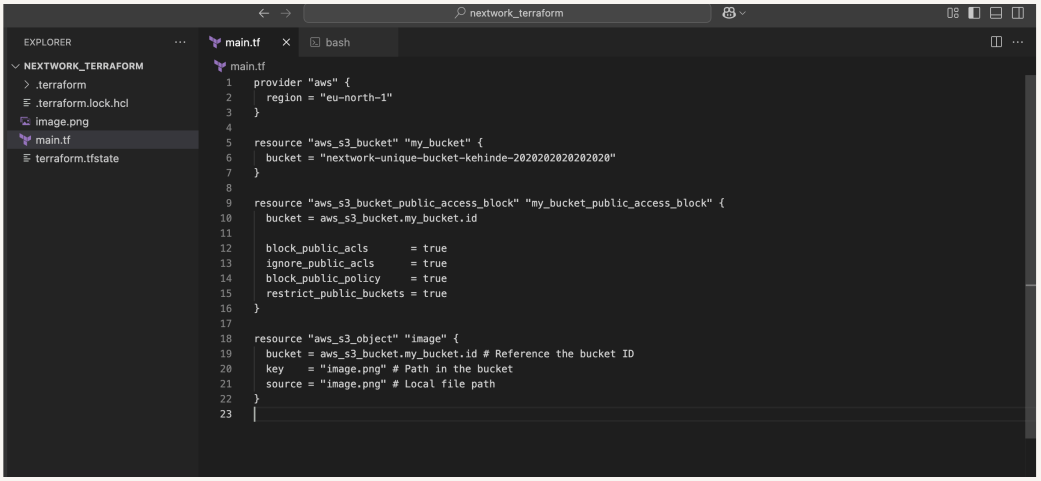

aws
Search
[Option+S]
Europe (Stockholm)
Account ID: 5112-3943-1868
kehindeabiuwa-IAM-Admin
Amazon S3
Buckets
Amazon S3
General purpose buckets
Directory buckets
Table buckets
Vector buckets
Access Grants
Access Points (General Purpose Buckets, FSx file systems)
Access Points (Directory Buckets)
Object Lambda Access Points
Multi-Region Access Points
Batch Operations
IAM Access Analyzer for S3
Block Public Access settings for this
General purpose buckets
All AWS Regions
Directory buckets
General purpose buckets (1) Info
Buckets are containers for data stored in S3.
Copy ARN
Empty
Delete
Create bucket
Find buckets by name
1
Name
AWS Region
Creation date
nextwork-unique-bucket-kehinde-2020202020202020
Europe (Stockholm) eu-north-1
August 26, 2025, 20:00:50 (UTC+01:00)
Account snapshot Info
Updated daily
View dashboard
Storage Lens provides visibility into storage usage and activity trends.
External access summary - new Info
Updated daily
External access findings help you identify bucket permissions that allow public access or access from other AWS accounts.

Uploading an S3 Object

I created a new resource block to upload a local file to my S3 bucket using: resource "aws_s3_object" "image" { bucket = aws_s3_bucket.my_bucket.id key = "image.png" # path/name in the bucket source = "image.png" # local file to upload # optional: # content_type = "image/png" } because I want Terraform to manage my content as code—each apply ensures the file exists in the bucket with the expected name and content, making deployments repeatable.

We need to run terraform apply again because we changed the configuration (added a new aws_s3_object to upload image.png). Terraform only makes real changes when you apply, so running it again will: Create the new object in the S3 bucket (and any other updates you added). Sync state so terraform.tfstate reflects the new resources. Best practice: run terraform plan first to preview the delta, then terraform apply to push it.

To validate that I've updated my configuration successfully, I opened the S3 console in my browser, navigated to my bucket, downloaded image.png, and confirmed it was the same image I had just uploaded—verifying the object upload worked as intended.


nextwork_terraform
EXPLORER
NEXTWORK_TERRAFORM
.terraform
.terraform.lock.hcl
image.png
main.tf
terraform.tfstate
main.tf
bash
main.tf
provider "aws" {
region = "eu-north-1"
}
resource "aws_s3_bucket" "my_bucket" {
bucket = "nextwork-unique-bucket-kehinde-2020202020202020"
}
resource "aws_s3_bucket_public_access_block" "my_bucket_public_access_block" {
bucket = aws_s3_bucket.my_bucket.id
block_public_acls = true
ignore_public_acls = true
block_public_policy = true
restrict_public_buckets = true
}
resource "aws_s3_object" "image" {
bucket = aws_s3_bucket.my_bucket.id # Reference the bucket ID
key = "image.png" # Path in the bucket
source = "image.png" # Local file path
}

The place to learn & showcase your skills

Check out nextwork.org for more projects