Week 4 Project Documentation - Secure Public Website with Private Infrastructure

Project Overview

In Weeks 1 through 3, I progressed from building a local on-premise server to manually deploying AWS infrastructure, and then automating that infrastructure using Terraform. Each week built upon the previous, establishing foundational knowledge of Linux servers, cloud networking, and Infrastructure as Code.

Week 4 represents the capstone project of this bootcamp and marks a significant shift in approach. This is no longer a solo learning exercise, this is a **real-world cloud migration executed as a professional engineering team**.

The Business Scenario

A European logistics company is migrating its on-premise public website to AWS. The company has strict requirements that reflect real enterprise security and operational standards:

- The website must be publicly accessible to customers
- Administrative and backend systems must remain completely private
- Only approved engineers with specific IAM roles may access servers
- Infrastructure must be fully automated using Terraform
- Monthly cloud spend must remain under $100
- Security controls must follow enterprise best practices, including:
    - No SSH ports exposed to the internet
    - Separation of public and private systems
    - Identity-based access control (not IP-based)
    - Centralized audit logging of all server access

A Solutions Architect has designed the high-level strategy. My team and I are the cloud engineers hired on contract to implement it correctly and securely.

What Makes Week 4 Different

This project introduces several production-level concepts and constraints:

Team Collaboration: Both team members deploy into one shared AWS environment. Only one person runs `terraform apply`. Coordination and communication are required.

Production Security Model: This architecture implements **defense in depth**:

- Public systems are isolated from private systems
- Access is controlled by IAM roles, not passwords or IP addresses
- AWS Systems Manager (SSM) replaces traditional SSH access
- A NAT Gateway allows private servers to download updates without being exposed to the internet

Manual + Automated Hybrid: IAM roles and access controls are created manually in the AWS Console (Part A), while all infrastructure is built using Terraform (Part B). This reflects real-world practice: security foundations are established first, automation follows.

Real Enterprise Requirements: This is not a sandbox. The architecture must be secure enough to handle real customer traffic, maintain uptime, and meet compliance standards.

Architecture Overview

The final system will consist of three tiers:

Public Tier (Presentation Layer):

- EC2 instance running Apache web server
- Lives in a public subnet with internet access
- Serves the public-facing website
- Loads static image assets from Amazon S3

Private Tier (Admin / Backend Layer):

- EC2 instance for administrative tasks
- Lives in a private subnet with no public IP address
- Cannot be reached directly from the internet
- Access only through AWS Systems Manager Session Manager

Static Asset Tier:

- Amazon S3 bucket hosting the website image
- Publicly accessible via object URL
- Decoupled from compute, enabling independent scaling

Key Concepts Introduced

AWS Systems Manager (SSM): A secure, auditable way to access servers without opening SSH ports. Access is controlled by IAM, not IP addresses.

NAT Gateway: Allows private servers to initiate outbound connections (for updates and patches) while remaining unreachable from the internet.

Public vs Private Subnets: Public subnets route traffic through an Internet Gateway. Private subnets route outbound traffic through a NAT Gateway but accept no inbound internet traffic.

IAM Role-Based Access: Engineers are granted specific levels of access (Admin, Operator, ReadOnly) through IAM roles, not through shared passwords or keys.

Break-Glass Access: A single IP address is allowed emergency SSH access as a fallback if Systems Manager fails.

Project Outcome

By the end of this project, I will have deployed a secure, production-style cloud architecture that demonstrates:

- Proper network segmentation (public/private isolation)
- Identity-based access control
- Infrastructure automation with Terraform
- Enterprise security practices that reduce attack surface
- Team collaboration in a shared cloud environment

This architecture proves that infrastructure is not just about making things work, it's about making things work **securely, predictably, and professionally**.

In Week 1, I learned how servers work. In Week 2, I learned how to build cloud infrastructure. In Week 3, I learned how to automate it. In Week 4, I'm learning how to **secure it like an enterprise**.

This project will be completed in three parts:

Part A — Manual IAM role creation in AWS Console (access control foundation)

Part B — Terraform deployment of VPC, subnets, EC2 instances, security groups, and NAT Gateway

Part C — Console configuration, Apache installation, website deployment, and SSM access testing

Part A — Manual Setup (AWS Console)

Step 1: Create IAM Role for EC2 (SSM Access)

What I did:

- Logged into AWS Console as my IAM admin user (bootcamp-admin)
- Navigated to **IAM → Roles → Create role**
- Selected trusted entity type: **AWS service**
- Selected use case: **EC2**
- Clicked **Next**
- On the permissions page, searched for and selected: **AmazonSSMManagedInstanceCore**
- Clicked **Next**
- Named the role: **Week4-EC2-SSM-Role**
- Reviewed the configuration:
    - Trusted entities: `ec2.amazonaws.com`
    - Attached policies: `AmazonSSMManagedInstanceCore`
- Clicked **Create role**
- Verified the role appeared in the IAM Roles list

What this did:

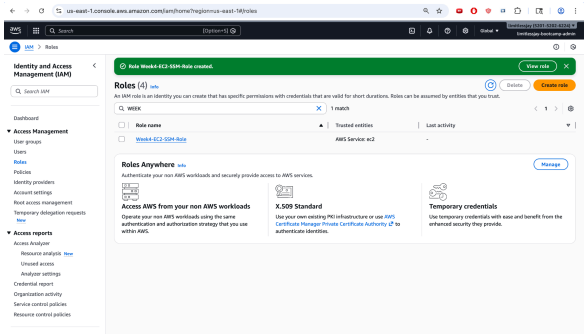
Created an IAM role that EC2 instances can assume to communicate with AWS Systems Manager. This role grants the EC2 instance permission to:

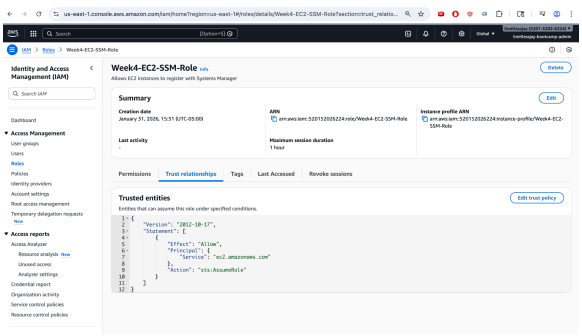
- Register itself with Systems Manager
- Send heartbeat signals and status updates
- Receive commands from Session Manager
- Allow authorized users to establish secure shell sessions through SSM (without opening SSH port 22)

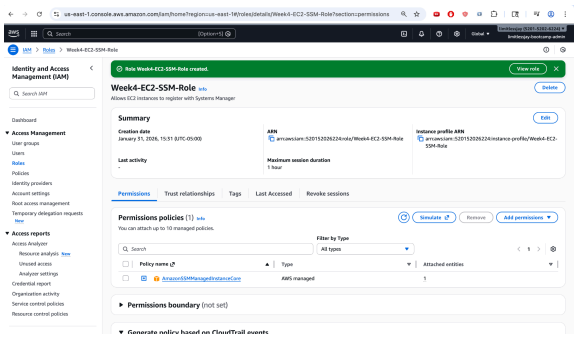
Why required:

AWS Systems Manager is a separate AWS service that requires explicit permission to interact with EC2 instances. Without this role:

- EC2 instances cannot register with SSM
- Session Manager connections will fail
- Engineers would be forced to use traditional SSH with open port 22 (security risk)
- There would be no centralized audit trail of server access
- This is how real companies secure server access at scale. Instead of managing SSH keys and opening ports, access is controlled through IAM and Session Manager, with every connection logged and auditable.
- This role establishes the foundation for secure, keyless server access in Part C.







Step 2: Create IAM Roles for Engineers

What I did:

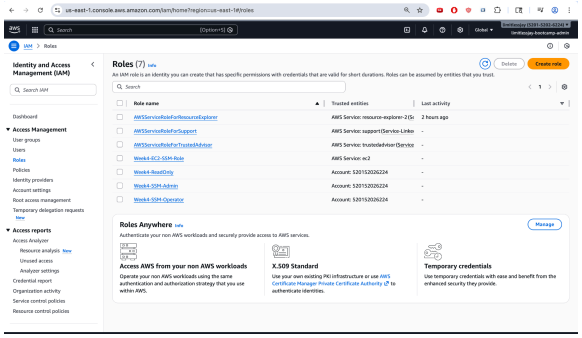
- Created three IAM roles for different engineer access levels:
  - **Week4-SSM-Admin**: Attached AmazonSSMFullAccess and AmazonEC2ReadOnlyAccess policies, trusted entity: AWS account
  - **Week4-SSM-Operator**: Created custom policy (Week4-SSM-Operator-Policy) with limited SSM permissions (ssm:StartSession, ssm:TerminateSession, ssm:DescribeSessions, ssm:DescribeInstanceInformation, ec2:DescribeInstances), then created role with custom policy attached, trusted entity: AWS account
  - **Week4-ReadOnly**: Attached ReadOnlyAccess policy, trusted entity: AWS account
- Verified all four roles (including Week4-EC2-SSM-Role) appeared in IAM Roles list

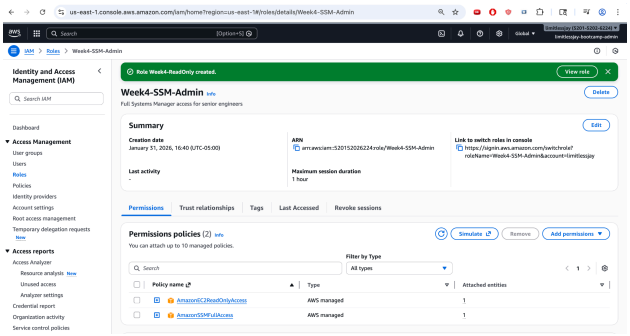
What this did:

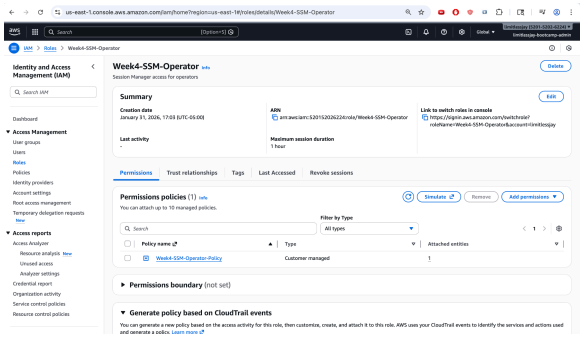
Created three distinct IAM roles implementing Role-Based Access Control (RBAC) for engineers. Week4-SSM-Admin grants senior engineers full Systems Manager access and read-only EC2 access. Week4-SSM-Operator grants day-to-day engineers permission to start and terminate Session Manager sessions with ability to view instances and active sessions, but no ability to change SSM configuration or modify EC2 instances. Week4-ReadOnly grants junior engineers and auditors view-only access to all AWS resources with no ability to modify anything or connect to servers. Note: ec2:DescribeInstances was included in the Operator policy to enable functional console experience for viewing available instances when starting sessions.

Why required:

The principle of least privilege requires each engineer receive only minimum permissions required for their role, reducing security risk and preventing accidental changes. Without role separation, everyone would need admin access (security risk), junior engineers could accidentally delete production resources, there would be no way to distinguish responsibility levels, and audit trails would be unclear. Real companies implement RBAC to ensure compliance with security frameworks (SOC 2, ISO 27001), protect production environments from accidental changes, create clear audit trails of who accessed what and when, and reduce blast radius when credentials are compromised. These roles will be tested in Part C after infrastructure deployment.







Step 3: Choose One SSH "Break-Glass" IP

What I did:

- Determined my public IPv4 address using https://whatismyipaddress.com/
- Recorded my IP address: `75.18.114.184`
- Converted to CIDR notation: `75.18.114.184/32`
- Documented this is my home network IP address (Fort Lauderdale, FL - AT&T Enterprises LLC)
- Confirmed this IP will be used as the single break-glass SSH access point
- Noted teammate will use SSM-only access (no SSH access)
- Date recorded: January 31, 2026

What this did:

Established a single designated IP address (75.18.114.184/32) that will be granted emergency SSH access to the public EC2 instance only. This creates a break-glass access mechanism - a backup entry point used only when the primary access method (AWS Systems Manager) is unavailable. The access model is: primary access for all engineers via Systems Manager Session Manager (no SSH, no open ports, IAM-controlled), break-glass access for one IP only via traditional SSH on port 22 restricted to 75.18.114.184/32, and no break-glass access for private instance since it has no public IP. This IP will be configured in the public security group rules during Part B Terraform deployment.

Why required:

Multiple layers of security require multiple layers of access - if the primary access method fails (Systems Manager service outage, SSM agent failure, IAM authentication issues), engineers need a backup method to troubleshoot and restore service. Restricting SSH to a single IP address (75.18.114.184/32) instead of the entire internet (0.0.0.0/0) reduces attack surface by 99.9999%. Attackers cannot exploit port 22 unless attacking from exactly this IP, which still requires possession of the private SSH key for additional defense in depth. This represents the trade-off of slightly increased attack surface (one open SSH port to one IP) for significantly increased operational resilience (ability to recover from SSM failures). This is standard enterprise practice, equivalent to data centers having physical override keys in addition to badge access.

Part B — Terraform (Infrastructure Only)

Step 4: Terraform Project Structure

What I did:

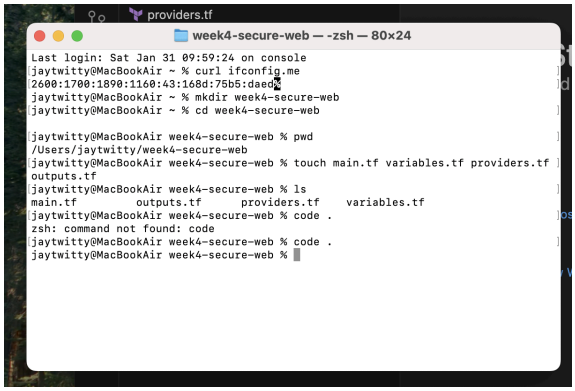
- Opened Terminal on local machine
- Created new project directory: `mkdir week4-secure-web` and `cd week4-secure-web`
- Created four Terraform configuration files: `touch main.tf variables.tf providers.tf outputs.tf`
- Verified files created using `ls`
- Opened project in Visual Studio Code: `code .`
- Confirmed all four files present and empty in VS Code editor

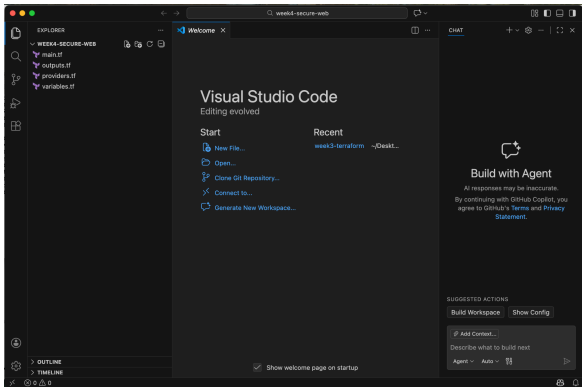
What this did:

Established a professional Terraform project structure with four separate configuration files. `main.tf` will contain infrastructure resource definitions (VPC, subnets, EC2 instances, security groups, NAT Gateway). `variables.tf` will contain input variable declarations (AWS region, break-glass IP). `providers.tf` will configure the AWS provider and authentication. `outputs.tf` will define information to display after deployment (public IPs, instance IDs). This separation of concerns follows Terraform best practices, making the codebase easier to read, simpler to modify, more maintainable for team collaboration, and aligned with industry standards.

Why required:

Enterprise infrastructure projects separate configuration by purpose, allowing engineers to quickly locate where to make changes, understand project structure at a glance, and collaborate without conflicts. While Terraform reads all `.tf` files regardless of naming, the four-file structure (main, variables, providers, outputs) is the industry standard used by HashiCorp, enterprise teams, open-source modules, and training programs. This structure provides maintainability - future engineers can immediately understand where infrastructure is defined, what can be customized, how authentication works, and what information deployment provides. This mirrors the Week 3 workflow, reinforcing professional Infrastructure as Code development practices.





Step 5: Provider Configuration

What I did:

- Opened `providers.tf` in Visual Studio Code
- Added AWS provider configuration: `provider "aws" { region = var.aws_region }`

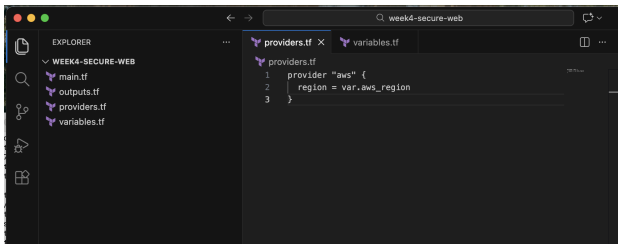
- Opened `variables.tf` in Visual Studio Code
- Defined two input variables:
    - `aws_region` with default value "us-east-2"
    - `allowed_ssh_cidr` with description "Single IP allowed SSH access" (no default)
- Saved both files

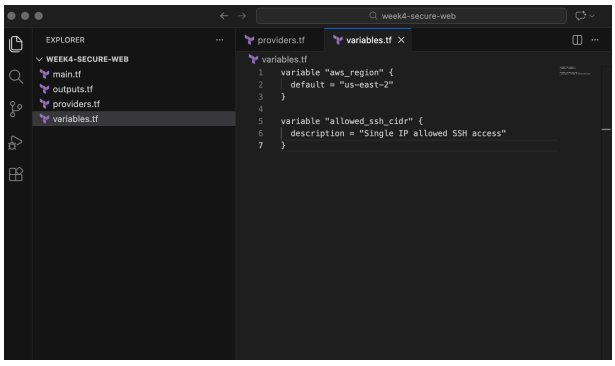
What this did:

Configured Terraform to use the AWS provider and set the deployment region using a variable reference. The `aws_region` variable defaults to "us-east-2" (Ohio) and is referenced by the provider configuration. The `allowed_ssh_cidr` variable defines the single IP address allowed SSH access (break-glass access) with no default value - it must be provided at runtime with value 75.18.114.184/32. Terraform automatically uses AWS credentials configured locally via `aws configure` from Week 3. No credentials are hardcoded in files, following security best practices.

Why required:

Terraform cannot interact with AWS without a provider, which translates Terraform syntax into AWS API calls, manages authentication, and handles resource lifecycle. All AWS resources must exist in a specific region - using a variable for the region provides flexibility, keeps configuration centralized, and follows Terraform best practices. Variables make infrastructure code flexible and reusable - the `aws_region` variable allows deployment to different regions without code modification, while `allowed_ssh_cidr` separates the break-glass IP from security group code, allowing team members to provide their own IPs without editing files. This configuration establishes the foundation for all infrastructure deployment, with the `allowed_ssh_cidr` variable referenced later when creating security group rules.





You're right! Let me match YOUR format:

Step 5B: Configure Remote State Storage (S3 Backend)

What I did:

- Created S3 bucket for Terraform state storage: aws s3api create-bucket --bucket week4-terraform-state-jaytwitty --region us-east-2 --create-bucket-configuration LocationConstraint=us-east-2
- Enabled versioning on state bucket: aws s3api put-bucket-versioning --bucket week4-terraform-state-jaytwitty --versioning-configuration Status=Enabled
- Created DynamoDB table for state locking: aws dynamodb create-table --table-name week4-terraform-locks --attribute-definitions AttributeName=LockID,AttributeType=S --key-schema AttributeName=LockID,KeyType=HASH --billing-mode PAY_PER_REQUEST --region us-east-2
- Opened providers.tf in Visual Studio Code
- Added Terraform backend configuration block at the top of file with S3 bucket name, state file key path, region, DynamoDB table name, and encryption enabled
- Saved the file
- Ran terraform init -migrate-state to migrate existing local state to S3
- Typed yes when prompted to copy state to new backend
- Verified state file uploaded to S3: aws s3 ls s3://week4-terraform-state-jaytwitty/week4-secure-web/

What this did:

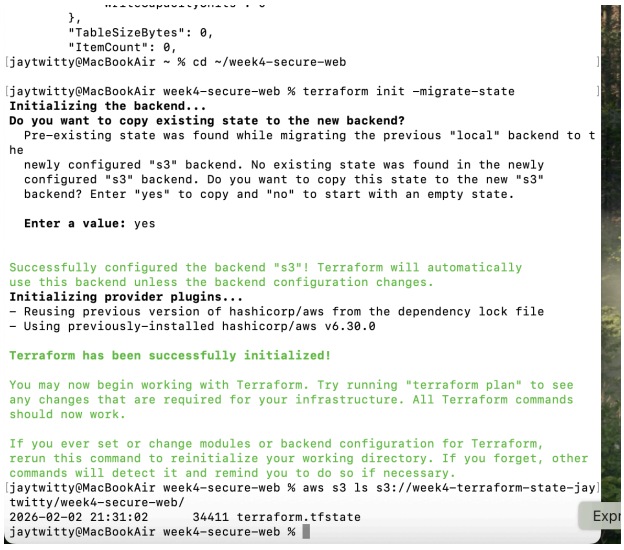
Configured Terraform to use remote state storage instead of local terraform.tfstate file. The S3 bucket stores the state file with versioning enabled for rollback capability. The DynamoDB table provides state locking to prevent concurrent modifications when multiple team members work on infrastructure simultaneously. The backend configuration in providers.tf specifies bucket location (week4-terraform-state-jaytwitty), state file path (week4-secure-web/terraform.tfstate), and enables server-side encryption. Running terraform init -migrate-state copied the existing local state to S3 and configured Terraform to use remote backend for all future operations. All subsequent Terraform commands now read and write state from S3 rather than local filesystem.

Why required:

Team collaboration requires shared infrastructure state that all members can access. Without remote state, each team member maintains a separate local terraform.tfstate file, causing conflicts, duplicate resources, and infrastructure drift. The S3 backend provides a single source of truth accessible to the entire team. State locking via DynamoDB prevents race conditions by ensuring only one person can modify infrastructure at a time - if two engineers run terraform

apply simultaneously, DynamoDB locks the state, allowing only one to proceed while the other waits. This prevents conflicting changes and corrupted state. Versioning enables recovery from mistakes or state corruption by allowing rollback to previous versions. Encryption protects sensitive data in the state file including resource IDs, IP addresses, and configuration values. The project specifications explicitly require "Terraform state stored remotely (S3 + DynamoDB)" and "Only one person runs terraform apply", both of which this configuration enables. This reflects real-world enterprise practices where infrastructure is managed by teams collaborating on shared codebases. Remote state also provides disaster recovery - if a team member's machine fails, the infrastructure state persists safely in S3 and can be accessed from any authorized machine.

```
Last login: Sun Feb  1 10:41:18 on ttys000
jaytwitty@MacBookAir ~ % aws s3api create-bucket \
  --bucket week4-terraform-state-jaytwitty \
  --region us-east-2 \
  --create-bucket-configuration LocationConstraint=us-east-2
{
    "Location": "http://week4-terraform-state-jaytwitty.s3.amazonaws.com/",
    "BucketArn": "arn:aws:s3:::week4-terraform-state-jaytwitty"
}
jaytwitty@MacBookAir ~ % aws s3api put-bucket-versioning \
  --bucket week4-terraform-state-jaytwitty \
  --versioning-configuration Status=Enabled
jaytwitty@MacBookAir ~ % aws dynamodb create-table \
  --table-name week4-terraform-locks \
  --attribute-definitions AttributeName=LockID,AttributeType=S \
  --key-schema AttributeName=LockID,KeyType=HASH \
  --billing-mode PAY_PER_REQUEST \
  --region us-east-2
{
    "TableDescription": {
        "AttributeDefinitions": [
            {
                "AttributeName": "LockID",
                "AttributeType": "S"
            }
        ],
        "TableName": "week4-terraform-locks",
        "KeySchema": [
            {
                "AttributeName": "LockID",
                "KeyType": "HASH"
            }
        ],
        "TableStatus": "CREATING",
        "CreationDateTime": "2026-02-02T21:23:48.411000-05:00",
        "ProvisionedThroughput": {
            "NumberOfDecreasesToday": 0,
            "ReadCapacityUnits": 0,
            "WriteCapacityUnits": 0
        },
        "TableSizeBytes": 0,
        "ItemCount": 0,
```



```
        },
        "TableSizeBytes": 0,
        "ItemCount": 0,
jaytwitty@MacBookAir ~ % cd ~/week4-secure-web

jaytwitty@MacBookAir week4-secure-web % terraform init -migrate-state
Initializing the backend...
Do you want to copy existing state to the new backend?
  Pre-existing state was found while migrating the previous "local" backend to t
he
  newly configured "s3" backend. No existing state was found in the newly
  configured "s3" backend. Do you want to copy this state to the new "s3"
  backend? Enter "yes" to copy and "no" to start with an empty state.

  Enter a value: yes


Successfully configured the backend "s3"! Terraform will automatically
use this backend unless the backend configuration changes.
Initializing provider plugins...
- Reusing previous version of hashicorp/aws from the dependency lock file
- Using previously-installed hashicorp/aws v6.30.0

Terraform has been successfully initialized!

You may now begin working with Terraform. Try running "terraform plan" to see
any changes that are required for your infrastructure. All Terraform commands
should now work.

If you ever set or change modules or backend configuration for Terraform,
rerun this command to reinitialize your working directory. If you forget, other
commands will detect it and remind you to do so if necessary.
jaytwitty@MacBookAir week4-secure-web % aws s3 ls s3://week4-terraform-state-jay
twitty/week4-secure-web/
2026-02-02 21:31:02      34411 terraform.tfstate
jaytwitty@MacBookAir week4-secure-web %
```

Step 6: Networking (VPC, Subnets, IGW)

What I did:

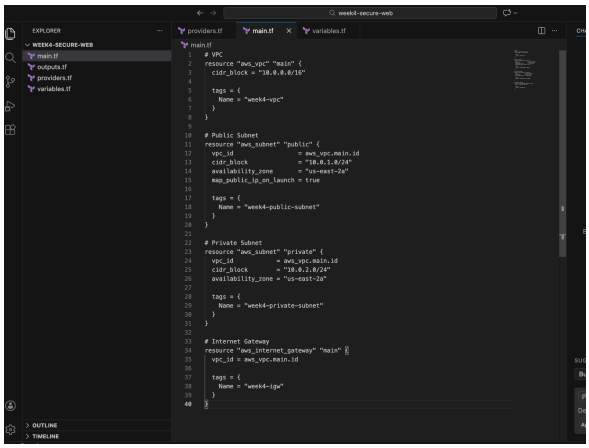
- Opened `main.tf` in Visual Studio Code
- Created VPC with CIDR block 10.0.0.0/16
- Created public subnet with CIDR block 10.0.1.0/24 in us-east-2a with `map_public_ip_on_launch = true`
- Created private subnet with CIDR block 10.0.2.0/24 in us-east-2a
- Created Internet Gateway and attached it to the VPC
- Saved the file

What this did:

Created the network foundation for the infrastructure. The VPC (10.0.0.0/16) provides an isolated network with 65,536 possible IP addresses. The public subnet (10.0.1.0/24) has 256 IPs and automatically assigns public IPs to resources, hosting the web server. The private subnet (10.0.2.0/24) has 256 IPs and does not assign public IPs, hosting the admin server. Both subnets are in availability zone us-east-2a for low latency. The Internet Gateway enables internet connectivity for resources in public subnets.

Why required:

Network segmentation separating public and private resources is fundamental security practice. The VPC creates a logically isolated network preventing interference with other AWS accounts. The public subnet design with automatic public IP assignment allows customer traffic to reach the website via Internet Gateway. The private subnet design without public IP assignment means the admin server cannot be reached from the internet and will route outbound traffic through NAT Gateway for updates only. This provides defense in depth - even if security groups are misconfigured, the private subnet has no direct internet route. Placing both subnets in the same availability zone keeps resources in the same physical data center for low latency.



Step 7: NAT Gateway

What I did:

- Opened `main.tf` in Visual Studio Code
- Created Elastic IP for NAT Gateway with `domain = "vpc"`
- Created NAT Gateway in public subnet with Elastic IP attached
- Saved the file

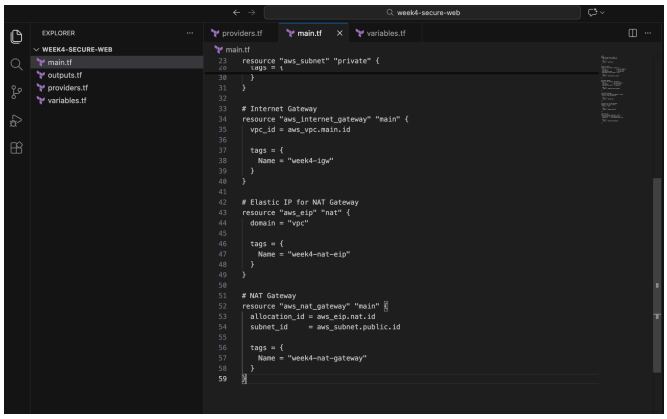
What this did:

Allocated a static public IP address (Elastic IP) and created a managed NAT Gateway service in the public subnet. The NAT Gateway provides outbound-only internet access for resources in the private subnet. When private servers initiate outbound connections (like `apt update`),

traffic routes to the NAT Gateway, which uses its public IP to make the request to the internet. The internet responds to the NAT Gateway, which forwards responses back to the private server. This creates a one-way traffic model - private servers can initiate outbound connections but the internet cannot initiate inbound connections to private servers.

Why required:

Private servers must never have public IP addresses or be directly reachable from the internet, but they still need to download operating system updates and security patches. The NAT Gateway solves this by providing outbound-only internet access while keeping private servers completely isolated from inbound internet traffic. The NAT Gateway must live in the public subnet because it needs internet access via the Internet Gateway to function as a middleman. Using an Elastic IP ensures a consistent outbound IP address for audit logs and persists even if the NAT Gateway is replaced. This enables the private subnet architecture pattern that separates public-facing resources from private backend systems, a fundamental security best practice. Note that NAT Gateway is one of the more expensive AWS services at approximately $0.045/hour plus $0.045/GB processed.



Step 8: Route Tables

What I did:

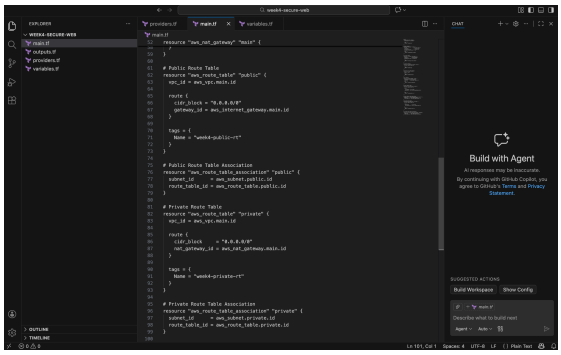
- Opened `main.tf` in Visual Studio Code
- Created public route table with routing rule: all internet-bound traffic (0.0.0.0/0) routes through Internet Gateway
- Associated public route table with public subnet
- Created private route table with routing rule: all internet-bound traffic (0.0.0.0/0) routes through NAT Gateway
- Associated private route table with private subnet
- Saved the file

What this did:

Created two route tables defining how traffic flows in and out of each subnet. The public route table directs all internet traffic (0.0.0.0/0) through the Internet Gateway, enabling bidirectional internet access for the public subnet. The private route table directs all internet traffic through the NAT Gateway, enabling outbound-only internet access for the private subnet. Route table associations linked each subnet to its corresponding route table. Public subnet traffic can now send and receive from the internet via the Internet Gateway, while private subnet traffic can only initiate outbound connections via NAT Gateway - inbound traffic from the internet is blocked because no route exists.

Why required:

Without route tables, subnets cannot communicate beyond the VPC. Route tables act as the GPS for network traffic, determining where packets go based on destination. Different route tables for public and private subnets enforce the security boundary - public gets bidirectional internet access (needed for web server), while private gets unidirectional access (outbound only for updates). Creating a route table isn't enough - it must be explicitly associated with a subnet or subnets would use the VPC's default route table. CIDR block 0.0.0.0/0 represents the default route, catching all internet-bound traffic and directing it to the appropriate gateway. This provides defense in depth - even if security group rules are misconfigured, the private route table ensures no inbound route exists from the internet to private resources.



Step 9: Security Groups (IMPORTANT)

What I did:

- Opened `main.tf` in Visual Studio Code
- Created public web security group with three rules:
    - HTTP (port 80) from anywhere (0.0.0.0/0)
    - SSH (port 22) from my break-glass IP only (var.allowed_ssh_cidr)
    - All outbound traffic allowed
- Created private admin security group with:
    - NO inbound rules (access via SSM only)
    - All outbound traffic allowed (for updates via NAT Gateway)

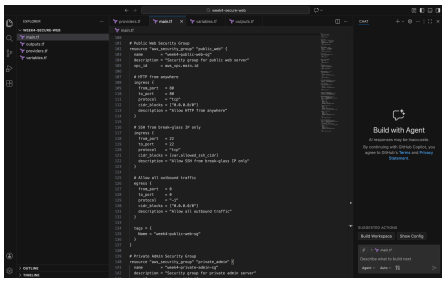
- Saved the file

What this did:

Created two security groups (virtual firewalls) enforcing network-level isolation between public and private tiers. The public web security group allows HTTP traffic from anywhere for customer website access, SSH from my break-glass IP only (75.18.114.184/32) for emergency access, and all outbound traffic for updates. The private admin security group has zero inbound rules - no ports open whatsoever - with all outbound traffic allowed for NAT Gateway updates. Access to the private server is only possible via AWS Systems Manager Session Manager, which doesn't require security group inbound rules because it uses AWS's internal network infrastructure and IAM authentication.

Why required:

Security groups enforce blast radius control - if the public web server is compromised, the attacker cannot pivot to the admin server because there is no network path. The private server has multiple defense layers: no public IP, no internet route, no inbound security group rules, and IAM-based access only. The break-glass SSH restriction on the public server (one IP only vs. the entire internet) reduces attack surface by 99.9999% while maintaining emergency access. Even though the private server lives in a private subnet with no public IP, enforcing zero inbound security group rules provides defense in depth, protects against accidental public IP assignment, satisfies compliance requirements, and future-proofs the configuration. This implements the principle of least privilege at the network firewall level, ensuring each server receives only the minimum traffic required for its function.



Step 10: Create EC2 Instances (Terraform)

What I did:

- Found Ubuntu Server 24.04 LTS AMI ID for us-east-2 region using AWS Console
- Recorded AMI ID: `ami-06e3c045d79fd65d9`
- Created SSH key pair for break-glass access: `ssh-keygen -t rsa -b 2048 -f ~/.ssh/week4-key`
- Opened `main.tf` in Visual Studio Code
- Added SSH key pair resource to upload public key to AWS

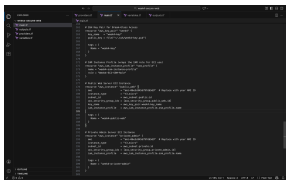
- Created IAM instance profile that wraps the Week4-EC2-SSM-Role
- Created public web server EC2 instance in public subnet
- Created private admin server EC2 instance in private subnet
- Saved the file

What this did:

Created two EC2 instances and supporting resources. Generated a 2048-bit RSA key pair locally (private key at `~/.ssh/week4-key`, public key uploaded to AWS as "week4-key") enabling break-glass SSH access to the public server. Created an IAM instance profile wrapping the Week4-EC2-SSM-Role, allowing both instances to communicate with Systems Manager. The public web server (Ubuntu 24.04, t3.micro) was placed in the public subnet with the public security group, SSH key attached, and will automatically receive a public IP. The private admin server (Ubuntu 24.04, t3.micro) was placed in the private subnet with the private security group, no SSH key, and no public IP. Both instances have the SSM role attached via instance profile.

Why required:

EC2 instances are the actual servers that run applications - without them, the infrastructure would exist but have nothing running on it. Ubuntu 24.04 LTS was chosen for long-term support and compatibility with Apache. The t3.micro instance type provides sufficient resources (1 vCPU, 1GB RAM) while staying within budget at approximately $7.50/month per instance if running 24/7. The public instance includes an SSH key for break-glass emergency access if Systems Manager fails, while the private instance excludes SSH entirely, forcing use of Systems Manager as the only access method. This demonstrates proper private server isolation. EC2 instances require an instance profile to use IAM roles, which provides temporary credentials and enables Session Manager access without opening network ports. This two-instance design implements defense in depth - the public server is exposed but isolated, while the private server remains completely unreachable from the internet.



Step 11: Define Outputs

Note on Project Instructions: The original project instructions list Step 11 as "Attach IAM Role to EC2 (Console)" - a manual step. However, since I'm using Terraform, the IAM role attachment was already completed in Step 10 when I defined the EC2 instances using the `iam_instance_profile` parameter. Therefore, I'm using Step 11 to define Terraform outputs instead.

What I did:

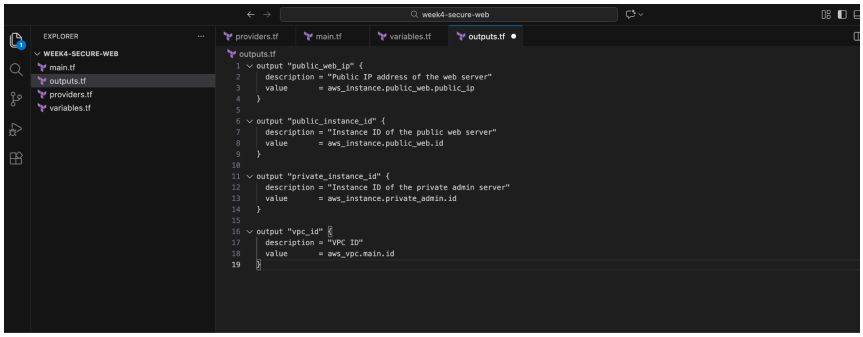
- Opened `outputs.tf` in Visual Studio Code
- Defined four output values to display after infrastructure deployment:
    - Public web server IP address
    - Public web server instance ID
    - Private admin server instance ID
    - VPC ID
- Saved the file

What this did:

Defined output values that Terraform displays after successful deployment, providing immediate access to critical information without requiring manual lookup in AWS Console. The `public_web_ip` output shows the public IP address needed to access the website. The `public_instance_id` and `private_instance_id` outputs provide the instance IDs required for Session Manager access and AWS CLI commands. The `vpc_id` output displays the VPC identifier for reference and troubleshooting. These outputs are displayed in the terminal immediately after `terraform apply` completes.

Why required:

Without outputs, finding critical information after deployment requires logging into AWS Console, navigating to services, and manually copying values. Outputs provide immediate terminal display of all essential information needed for Part C tasks. The outputs are automation-friendly and can be captured in shell scripts, used in CI/CD pipelines, or referenced by other tools. This is standard practice in professional Terraform projects because it makes deployment information programmatically accessible, reduces human error, and provides a clear interface for interacting with deployed resources. The IAM role attachment mentioned in original instructions was handled automatically in Step 10 via Terraform, which is superior to manual attachment because it's automated, repeatable, and documented in code.



Step 12: Deploy Infrastructure with Terraform

What I did:

- Opened Terminal and navigated to project directory: `cd ~/week4-secure-web`

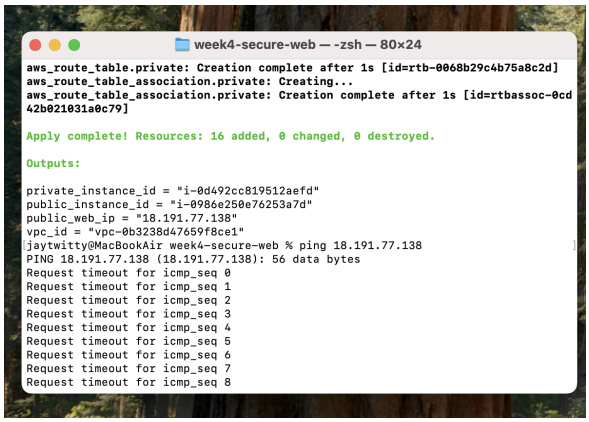
- Verified Terraform files present using `ls *.tf`
- Ran `terraform init` to initialize Terraform and download AWS provider plugin
- Ran `terraform plan` to preview infrastructure changes
    - Provided break-glass IP when prompted: `75.18.114.184/32`
    - Reviewed plan output showing 16 resources to be created
- Ran `terraform apply` to create infrastructure
    - Provided break-glass IP when prompted: `75.18.114.184/32`
    - Confirmed deployment by typing `yes`
    - Waited approximately 3 minutes for deployment to complete
- Recorded output values:
    - public_web_ip: `18.191.77.138`
    - public_instance_id: `i-0986e250e76253a7d`
    - private_instance_id: `i-0d492cc819512aefd`
    - vpc_id: `vpc-0b3238d47659f8ce1`
- Verified infrastructure in AWS Console confirming VPC exists in us-east-2, both EC2 instances running, and security groups attached
- Tested basic connectivity by pinging public web server IP (received timeout as expected due to security group blocking ICMP)

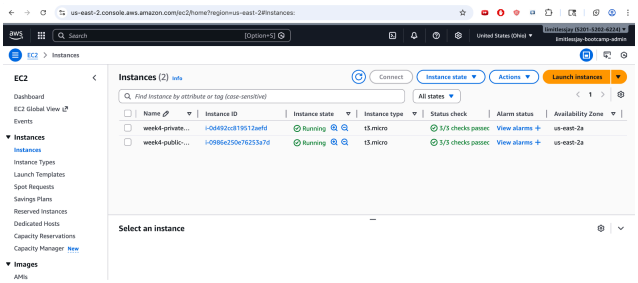
What this did:

Created all 16 AWS resources defined in Terraform code and deployed the complete infrastructure in us-east-2 region. `terraform init` downloaded the AWS provider plugin and initialized the working directory. `terraform plan` calculated dependencies and generated an execution plan showing 16 resources to add. `terraform apply` created resources in correct dependency order: VPC, Internet Gateway, subnets, Elastic IP, NAT Gateway (took ~90 seconds), route tables, security groups, SSH key pair, IAM instance profile, and both EC2 instances. Total deployment time was approximately 3 minutes with zero errors. The ping timeout was expected and normal - the security group only allows HTTP and SSH, blocking ICMP protocol, proving security rules work correctly.

Why required:

Writing Terraform code defines infrastructure declaratively but doesn't create anything. Deployment translates code into actual AWS resources following Infrastructure as Code best practices. The three-step process (init, plan, apply) is essential: init downloads provider plugins needed to communicate with AWS, plan validates configuration and shows what will be created to prevent costly mistakes, and apply creates resources in correct dependency order automatically. Terraform's state management tracks all created resources, enables future updates or destruction, and ensures idempotency - running apply again with no changes results in no modifications. This automated approach is superior to manual creation because it handles dependencies correctly, provides rollback capability, and enables version control of infrastructure.





PART C — CONSOLE CONFIGURATION & WEBSITE SETUP

Step 13: SSH Into Public EC2 (Break-Glass)

What I did:

- Verified SSH private key exists at `~/.ssh/week4-key`
- Set correct permissions on key file: `chmod 400 ~/.ssh/week4-key`
- Retrieved public web server IP from terraform outputs: `18.191.77.138`
- Connected to public web server via SSH: `ssh -i ~/.ssh/week4-key ubuntu@18.191.77.138`
- Accepted host key fingerprint on first connection by typing `yes`
- Verified successful connection by observing Ubuntu welcome message and command prompt
- Confirmed location by running:
    - `hostname` - showed `ip-10-0-1-xxx` (private DNS name)
    - `whoami` - showed `ubuntu` (logged in user)
    - `curl http://169.254.169.254/latest/meta-data/instance-id` - showed `i-0986e250e76253a7d` (instance ID)
- Explored system by checking home directory contents and OS version
- Remained connected for next step (Apache installation)

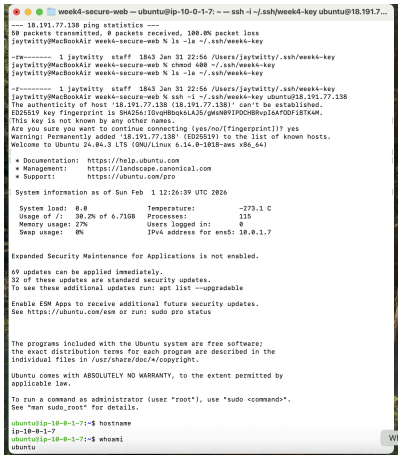
What this did:

Established an SSH (Secure Shell) connection to the public web server using break-glass access credentials. Used private key authentication instead of password, connecting as user

`ubuntu` (default for Ubuntu AMIs). The connection went through port 22, which is allowed by the security group for my IP only (75.18.114.184/32). SSH client saved the server's fingerprint to `~/.ssh/known_hosts` for future verification. With SSH access established, I now have command-line control to install software, configure settings, and manage the server.

Why required:

This step demonstrates the emergency backup access method configured in Part A, Step 3. While AWS Systems Manager Session Manager is the primary access method, SSH provides a fallback if SSM fails. Successfully connecting via SSH proves the public web security group is configured correctly - port 22 is open to my IP only, blocking all other IPs. SSH key-based authentication is more secure than passwords because the private key never transmits over the network and uses 2048-bit RSA encryption. The `chmod 400` command ensures proper key permissions, which SSH requires as a security feature. This break-glass access allows emergency server management when primary access methods are unavailable.



Step 14: Install Apache

What I did:

- Verified SSH connection to public web server
- Updated package lists: `sudo apt update`
- Installed Apache: `sudo apt install apache2 -y`
- Verified installation: `apache2 -v` showed Apache/2.4.58
- Checked status: `sudo systemctl status apache2` showed active (running) and enabled
- Tested locally: `curl localhost` returned HTML with "It works!"
- Tested from browser: navigated to http://18.191.77.138 and saw Apache default page
- Verified auto-start: `sudo systemctl is-enabled apache2` returned enabled

What this did:

Installed Apache HTTP Server version 2.4.58 on the public web server. Apache is now running and listening on port 80 for HTTP requests. The installation created the default website directory at `/var/www/html/` with a default index.html page. Apache is configured to start automatically on server reboot. Testing confirmed Apache responds to both local requests (from inside the server) and internet requests (from browser), proving the complete public tier architecture is functioning correctly including security group rules allowing HTTP traffic on port 80.

Why required:

Without a web server, the EC2 instance cannot serve web pages. Apache transforms the instance into a functioning web server capable of responding to HTTP requests from the internet. The security group allows port 80 traffic, but without Apache listening on that port, browsers would receive connection timeouts. Apache is the industry-standard web server powering approximately 30% of all websites globally. Installing Apache with systemd service management ensures it runs reliably and recovers automatically from server reboots. The successful browser test confirms end-to-end connectivity from internet through Internet Gateway, security group, and to Apache, validating the entire public website infrastructure.

Step 15: Create S3 Bucket & Upload Image

What I did:

- Logged into AWS Console as IAM admin user
- Navigated to S3 service
- Created new S3 bucket named `week4-logistics-website-jaytwitty` in us-east-2 region
- Unchecked "Block all public access" and acknowledged the warning
- Created the bucket successfully
- Uploaded image file `european_cargo_logistics_gmbh_cover.jpeg` to the bucket
- Attempted to make object public via ACL but Edit button was greyed out due to "Bucket Owner Enforced" setting

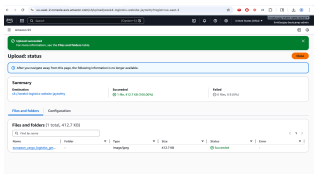
- Used alternative method: created bucket policy to make all objects publicly readable
- Navigated to bucket Permissions tab → Bucket policy → Edit
- Added policy allowing public read access (s3:GetObject) for all objects in bucket
- Saved bucket policy successfully
- Copied object URL: `https://week4-logistics-website-jaytwitty.s3.amazonaws.com/european_cargo_logistics_gmbh_cover.jpeg`
- Tested URL in browser and confirmed image displays publicly

What this did:

Created an Amazon S3 bucket to store static website assets separately from the EC2 web server. Uploaded a company logo image and configured public access using a bucket policy instead of ACL permissions. The bucket policy grants read access to all objects in the bucket, allowing anyone with the URL to view the images. This provides a publicly accessible object URL that can be referenced in website HTML.

Why required:

S3 provides durable, scalable storage for static website assets. Storing images in S3 instead of on the EC2 instance provides separation of concerns, resilience (images persist if EC2 is rebuilt), cost-effectiveness (S3 storage is cheaper), and performance (S3 is optimized for file delivery). Making objects publicly readable is necessary for website functionality because browsers must be able to retrieve images when loading the webpage. The object URL will be used in Step 16 when creating the website HTML.



Step 16: Create Website HTML

What I did:

- Verified SSH connection to public web server
- Opened Apache homepage file: `sudo nano /var/www/html/index.html`
- Deleted all existing content using Ctrl+K
- Created new HTML with company title, headings, and S3 image reference
- Used S3 URL in img tag: `https://week4-logistics-website-jaytwitty.s3.amazonaws.com/european_cargo_logistics_gmbh_cover.jpeg`
- Saved file with Ctrl+O, Enter
- Exited nano with Ctrl+X

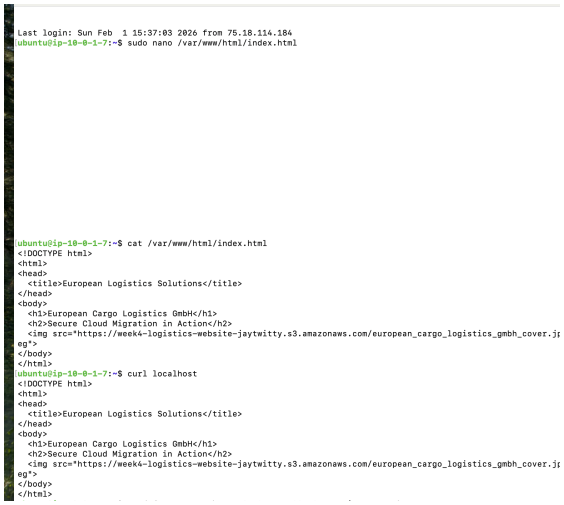
- Verified file with `cat /var/www/html/index.html`
- Tested locally with `curl localhost`
- Tested from browser at http://18.191.77.138
- Website displayed successfully with headings and S3 image

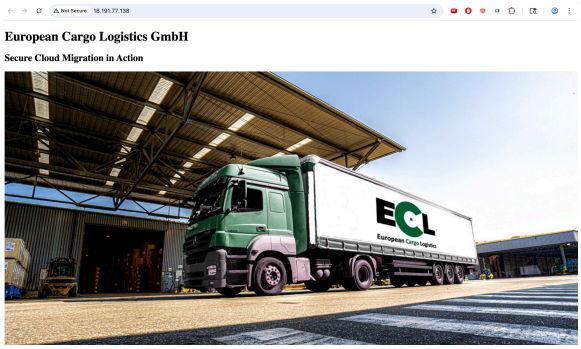
What this did:

Replaced Apache's default homepage with custom HTML for the European logistics company website. Apache serves the HTML from `/var/www/html/index.html` while the image loads from S3, demonstrating separation between compute (EC2) and storage (S3) tiers.

Why required:

Creating custom website content demonstrates the complete web hosting workflow. The multi-tier architecture is proven functional: Internet Gateway routes traffic, security group allows HTTP, Apache serves HTML, and S3 delivers static assets. This fulfills the project requirement of deploying a public-facing logistics company website using enterprise cloud architecture.





Step 17: Access Private Server via SSM

What I did:

- Logged into AWS Console as IAM admin user
- Navigated to Systems Manager service
- Opened Session Manager from Node Management section
- Clicked Start session button
- Selected private admin instance from list (Instance ID: `i-0d492cc819512aefd`, Name: `week4-private-admin`)
- Started session - browser opened new tab with shell prompt

- Verified connection by running `hostname` - showed `ip-10-0-2-xxx` confirming private subnet
- Checked for public IP with `curl ifconfig.me` - returned NAT Gateway's public IP, not instance IP
- Attempted metadata query for instance ID - no output (expected behavior in some SSM sessions)
- Checked network interface with `ip addr show` - showed only private IP (10.0.2.xxx), no public IP
- Tested internet access with `sudo apt update` - successfully downloaded package lists via NAT Gateway
- Terminated session from browser

What this did:

Established secure browser-based shell access to the private admin server using AWS Systems Manager Session Manager without requiring SSH, open ports, or SSH keys. Successfully connected to the instance despite it having no public IP address and zero inbound security group rules.

Verification confirmed the private server architecture is working correctly: the instance has only a private IP address (10.0.2.xxx), cannot be reached from the internet, but can access the internet for updates through the NAT Gateway. The Week4-EC2-SSM-Role (attached via Terraform in Part B, Step 10) enabled Session Manager connectivity by allowing the instance to register with Systems Manager.

Why required:

This step proves the enterprise security model is fully operational. The private admin server demonstrates defense-in-depth security: no public IP address, no SSH access, zero inbound security group rules, and access only via IAM-authenticated Session Manager. This architecture prevents direct internet attacks while maintaining administrative access through AWS's secure infrastructure.

Session Manager provides superior security compared to traditional SSH because it requires no open network ports, uses IAM for authentication and authorization, logs all session activity to CloudTrail for audit compliance, and routes connections through AWS's internal network rather than the public internet.

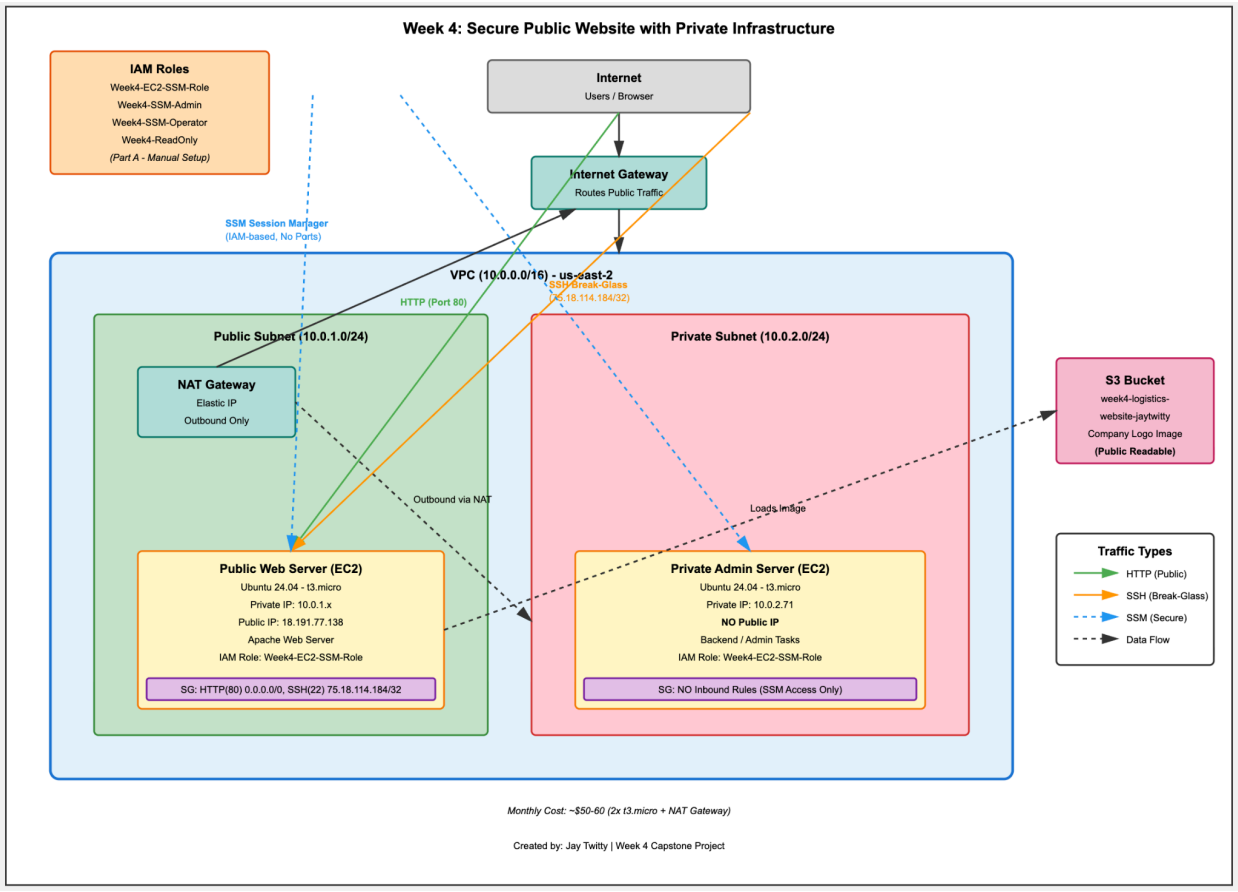
The successful connection validates that all security components are working together: IAM role attached to instance, Systems Manager agent running, IAM permissions allowing session creation, and network routing configured correctly. The ability to run `apt update` proves the NAT Gateway is functioning, allowing the private server to download security patches while remaining unreachable from the internet.

This architecture fulfills the project requirement of separating public and private infrastructure tiers with appropriate access controls for each, demonstrating production-grade cloud security practices.

```
$ hostname
ip-10-0-2-71
$ curl iconfig.me
$ curl ifconfig.me
3.132.239.108$ curl http://169.254.169.254/latest/meta-data/instance-id
$ curl http://169.254.169.254/latest/meta-data/instance-id
$
$ ip addr show
1: lo: <LOOPBACK,UP,LOWER_UP> mtu 65536 qdisc noqueue state UNKNOWN group default qlen 1000
    link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
    inet 127.0.0.1/8 scope host lo
       valid_lft forever preferred_lft forever
    inet6 ::1/128 scope host noprefixroute
       valid_lft forever preferred_lft forever
2: ens5: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 9001 qdisc mq state UP group default qlen 1000
    link/ether 02:9a:8b:9b:54:cb brd ff:ff:ff:ff:ff:ff
    altname enp0s5
    inet 10.0.2.71/24 metric 100 brd 10.0.2.255 scope global dynamic ens5
       valid_lft 1940sec preferred_lft 1940sec
    inet6 fe80::9a:8bff:fe9b:54cb/64 scope link
       valid_lft forever preferred_lft forever
$ sudo apt update
Hit:1 http://us-east-2.ec2.archive.ubuntu.com/ubuntu noble InRelease
Get:2 http://us-east-2.ec2.archive.ubuntu.com/ubuntu noble-updates InRelease [126 kB]
Get:3 http://us-east-2.ec2.archive.ubuntu.com/ubuntu noble-backports InRelease [126 kB]
Get:4 http://security.ubuntu.com/ubuntu noble-security InRelease [126 kB]
Get:5 http://us-east-2.ec2.archive.ubuntu.com/ubuntu noble-updates/main amd64 Packages [1717 kB]
Get:6 http://us-east-2.ec2.archive.ubuntu.com/ubuntu noble-updates/main amd64 Components [175 kB]
Get:7 http://us-east-2.ec2.archive.ubuntu.com/ubuntu noble-updates/universe amd64 Packages [1526 kB]
Get:8 http://us-east-2.ec2.archive.ubuntu.com/ubuntu noble-updates/universe amd64 Components [386 kB]
Get:9 http://us-east-2.ec2.archive.ubuntu.com/ubuntu noble-updates/restricted amd64 Components [212 B]
Get:10 http://us-east-2.ec2.archive.ubuntu.com/ubuntu noble-updates/multiverse amd64 Packages [32.1 kB]
Get:11 http://us-east-2.ec2.archive.ubuntu.com/ubuntu noble-updates/multiverse amd64 Components [940 B]
Get:12 http://us-east-2.ec2.archive.ubuntu.com/ubuntu noble-backports/main amd64 Components [7284 B]
Get:13 http://us-east-2.ec2.archive.ubuntu.com/ubuntu noble-backports/universe amd64 Components [10.5 kB]
Get:14 http://us-east-2.ec2.archive.ubuntu.com/ubuntu noble-backports/restricted amd64 Components [216 B]
Get:15 http://us-east-2.ec2.archive.ubuntu.com/ubuntu noble-backports/multiverse amd64 Components [212 B]
Get:16 http://security.ubuntu.com/ubuntu noble-security/main amd64 Components [21.5 kB]
Get:17 http://security.ubuntu.com/ubuntu noble-security/universe amd64 Components [74.2 kB]
Get:18 http://security.ubuntu.com/ubuntu noble-security/restricted amd64 Components [212 B]
Get:19 http://security.ubuntu.com/ubuntu noble-security/multiverse amd64 Components [212 B]
```

Session ID: limitlessjay-bootcamp-admin-sjjnh5onffo3hcjoqn3a9qu68a Shortcuts Instance ID: i-0d492cc819512aefd

```
$
$ ip addr show
1: lo: <LOOPBACK,UP,LOWER_UP> mtu 65536 qdisc noqueue state UNKNOWN group default qlen 1000
    link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
    inet 127.0.0.1/8 scope host lo
       valid_lft forever preferred_lft forever
    inet6 ::1/128 scope host noprefixroute
       valid_lft forever preferred_lft forever
2: ens5: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 9001 qdisc mq state UP group default qlen 1000
    link/ether 02:9a:8b:9b:54:cb brd ff:ff:ff:ff:ff:ff
    altname enp0s5
    inet 10.0.2.71/24 metric 100 brd 10.0.2.255 scope global dynamic ens5
       valid_lft 1940sec preferred_lft 1940sec
    inet6 fe80::9a:8bff:fe9b:54cb/64 scope link
       valid_lft forever preferred_lft forever
$ sudo apt update
Hit:1 http://us-east-2.ec2.archive.ubuntu.com/ubuntu noble InRelease
Get:2 http://us-east-2.ec2.archive.ubuntu.com/ubuntu noble-updates InRelease [126 kB]
Get:3 http://us-east-2.ec2.archive.ubuntu.com/ubuntu noble-backports InRelease [126 kB]
Get:4 http://security.ubuntu.com/ubuntu noble-security InRelease [126 kB]
Get:5 http://us-east-2.ec2.archive.ubuntu.com/ubuntu noble-updates/main amd64 Packages [1717 kB]
Get:6 http://us-east-2.ec2.archive.ubuntu.com/ubuntu noble-updates/main amd64 Components [175 kB]
Get:7 http://us-east-2.ec2.archive.ubuntu.com/ubuntu noble-updates/universe amd64 Packages [1526 kB]
Get:8 http://us-east-2.ec2.archive.ubuntu.com/ubuntu noble-updates/universe amd64 Components [386 kB]
Get:9 http://us-east-2.ec2.archive.ubuntu.com/ubuntu noble-updates/restricted amd64 Components [212 B]
Get:10 http://us-east-2.ec2.archive.ubuntu.com/ubuntu noble-updates/multiverse amd64 Packages [32.1 kB]
Get:11 http://us-east-2.ec2.archive.ubuntu.com/ubuntu noble-updates/multiverse amd64 Components [940 B]
Get:12 http://us-east-2.ec2.archive.ubuntu.com/ubuntu noble-backports/main amd64 Components [7284 B]
Get:13 http://us-east-2.ec2.archive.ubuntu.com/ubuntu noble-backports/universe amd64 Components [10.5 kB]
Get:14 http://us-east-2.ec2.archive.ubuntu.com/ubuntu noble-backports/restricted amd64 Components [216 B]
Get:15 http://us-east-2.ec2.archive.ubuntu.com/ubuntu noble-backports/multiverse amd64 Components [212 B]
Get:16 http://security.ubuntu.com/ubuntu noble-security/main amd64 Components [21.5 kB]
Get:17 http://security.ubuntu.com/ubuntu noble-security/universe amd64 Components [74.2 kB]
Get:18 http://security.ubuntu.com/ubuntu noble-security/restricted amd64 Components [212 B]
Get:19 http://security.ubuntu.com/ubuntu noble-security/multiverse amd64 Components [212 B]
Fetched 4330 kB in 1s (3475 kB/s)
Reading package lists... Done
Building dependency tree... Done
Reading state information... Done
69 packages can be upgraded. Run 'apt list --upgradable' to see them.
$
```



Step 18: Test IAM Role Permissions

What I did:

- **Tested Week4-SSM-Admin role:**
    - Switched to Week4-SSM-Admin role in AWS Console
    - Verified EC2 read-only access - could view instances but no Stop/Start options visible
    - Tested Systems Manager Session Manager access - successfully connected to private admin instance (i-0d492cc819512aefd)
    - Switched back to normal user

- **Tested Week4-SSM-Operator role:**
    - Switched to Week4-SSM-Operator role in AWS Console
    - Verified EC2 instance viewing worked (ec2:DescribeInstances permission functional)
    - Attempted Session Manager access - received error: "not authorized to perform: ssm:DescribeInstanceProperties"
    - Unable to access Session Manager UI despite having ssm:StartSession permission
    - Identified issue: Custom policy missing ssm:DescribeInstanceProperties and ec2:DescribeVolumes permissions
    - Switched back to normal user

- **Tested Week4-ReadOnly role:**
    - Switched to Week4-ReadOnly role in AWS Console
    - Verified EC2 viewing worked - saw both instances
    - Tested Stop instance action - UI showed option but execution failed with error: "not authorized to perform: ec2:StopInstances"
    - Tested Session Manager access - no instances visible, cannot connect
    - Verified IAM viewing worked - successfully viewed all four roles
    - Switched back to normal user

What this did:

Validated three-tier RBAC implementation through actual permission testing. Admin role worked correctly with full SSM access and EC2 read-only. **Operator role testing revealed a gap in the custom policy - while the project instructions specified ssm:StartSession, ssm:TerminateSession, ssm:DescribeSessions, ssm:DescribeInstanceInformation, and ec2:DescribeInstances, the role is missing ssm:DescribeInstanceProperties (required for Session Manager UI to list instances) and ec2:DescribeVolumes (required for viewing complete instance details). This prevents the Operator role from functioning as intended despite having core session permissions.** ReadOnly role demonstrated proper

least-privilege. AWS Console displayed all UI options but IAM enforcement blocked execution attempts. Testing proved IAM policies enforce access control independent of UI presentation.

Why required:

IAM role testing validates security controls before production use and reveals permission gaps. The project explicitly requires testing roles after EC2 deployment. **This testing discovered that incomplete IAM policies can break functionality even when core permissions exist - missing ssm:DescribeInstanceProperties prevents the entire Session Manager workflow despite having ssm:StartSession. This demonstrates the importance of thorough testing and shows ability to diagnose IAM permission errors by reading AWS error messages.** Testing proved the principle of least privilege works correctly - each role can only perform intended actions, with IAM blocking unauthorized attempts regardless of UI display. Real-world IAM troubleshooting requires reading error messages to identify missing permissions. **The Operator role findings represent authentic cloud engineering experience where documentation may be incomplete and testing reveals issues that need resolution.**