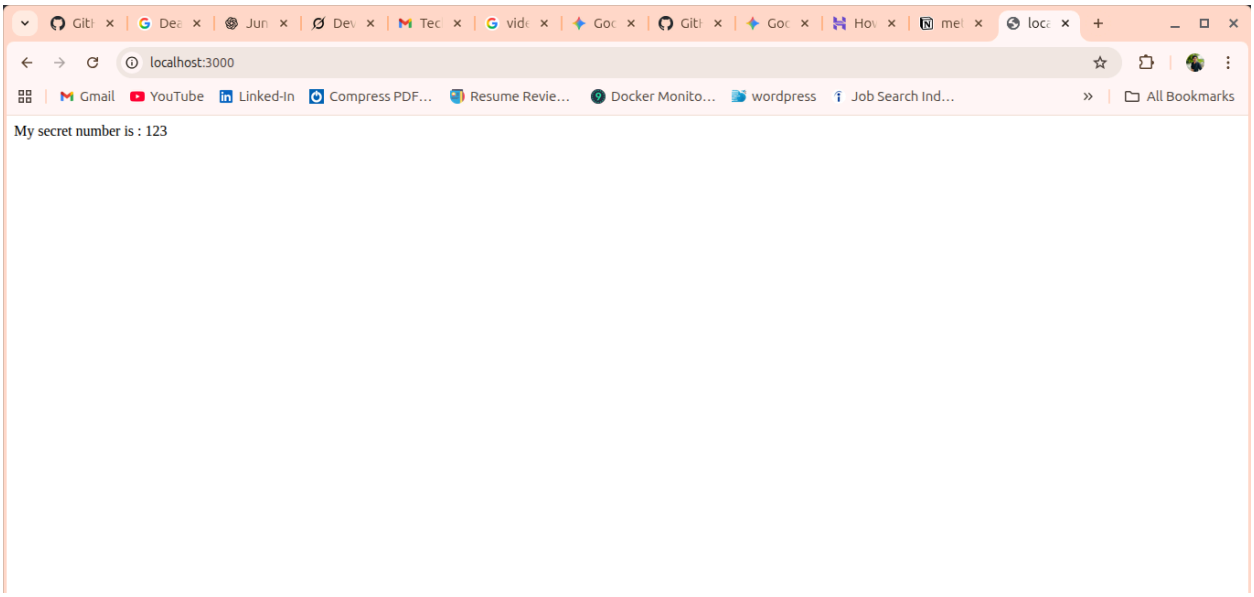
meta machine test

```
# Use an official Node.js runtime as the base image
FROM node:18-alpine

# Set working directory
WORKDIR /usr/src/app

# Copy package.json and package-lock.json
COPY package*.json ./

# Install dependencies
RUN npm install

# Copy the rest of the application
COPY . .

# Expose the app port (dotenv app doesn't specify one by default,
# but most Node apps listen on 3000 or 5000 — adjust if needed)
EXPOSE 3000

# Default command to run the app
CMD ["node", "index.js"]
```

docker build -t nodejs-dotenv .

ocker run --env-file .env -p 3000:3000 nodejs-dotenv




```
shefeek@mail: ~/Desktop/nodejs_dotenv
=> => sha256:f18232174bc91741fdf3da96d85011092101a032a93a388b79e99e69c2d5c870 3.64MB / 3.64MB
=> => sha256:dd71dde834b5c203d162902e6b8994cb2309ae049a0eabc4efea161b2b5a3d0e 40.01MB / 40.01MB
=> => sha256:1e5a4c89cee5c0826c540ab06d4b6b491c96eda01837f430bd47f0d26702d6e3 1.26MB / 1.26MB
=> => sha256:8d6421d663b4c28fd3ebc498332f249011d118945588d0a35cb9bc4b8ca09d9e 7.67kB / 7.67kB
=> => sha256:929b04d7c782f04f615cf785488fed452b6569f87c73ff666ad553a7554f0006 1.72kB / 1.72kB
=> => sha256:ee77c6cd7c1886ecc802ad6cedef3a8ec1ea27d1fb96162bf03dd3710839b8da 6.18kB / 6.18kB
=> => extracting sha256:f18232174bc91741fdf3da96d85011092101a032a93a388b79e99e69c2d5c870
=> => sha256:25ff2da83641908f65c3a74d80409d6b1b62ccfaab220b9ea70b80df5a2e0549 446B / 446B
=> => extracting sha256:dd71dde834b5c203d162902e6b8994cb2309ae049a0eabc4efea161b2b5a3d0e
=> => extracting sha256:1e5a4c89cee5c0826c540ab06d4b6b491c96eda01837f430bd47f0d26702d6e3
=> => extracting sha256:25ff2da83641908f65c3a74d80409d6b1b62ccfaab220b9ea70b80df5a2e0549
=> [2/5] WORKDIR /usr/src/app
=> [3/5] COPY package*.json ./
=> [4/5] RUN npm install
=> [5/5] COPY . .
=> exporting to image
=> => exporting layers
=> => writing image sha256:99ad1c68d14ab06f0e3780dae199ef30c7fdf52d377e6df1298f8a07381e73e2
=> => naming to docker.io/library/nodejs-dotenv
hefeek@mail:~/Desktop/nodejs_dotenv$ ls
ockerfile  env.example  index.js  package.json  package-lock.json  readme.md
hefeek@mail:~/Desktop/nodejs_dotenv$ touch .dockerignore
hefeek@mail:~/Desktop/nodejs_dotenv$ vim .dockerignore
hefeek@mail:~/Desktop/nodejs_dotenv$ cat .env
Y_SECRET_NUMBER=123
hefeek@mail:~/Desktop/nodejs_dotenv$ cat .dockerignore
*env
hefeek@mail:~/Desktop/nodejs_dotenv$ ocker run --env-file .env -p 3000:3000 nodejs-dotenv
ommand 'ocker' not found, did you mean:
 command 'docker' from snap docker (28.4.0)
 command 'docker' from deb docker.io (28.2.2-0ubuntu1~24.04.1)
 command 'docker' from deb podman-docker (4.9.3+ds1-1ubuntu0.2)
ee 'snap info <snapname>' for additional versions.
hefeek@mail:~/Desktop/nodejs_dotenv$ docker run --env-file .env -p 3000:3000 nodejs-dotenv
xample app listening on port 3000!
```

docker tag myapp:latest shefeekar/myapp:latest

shefeek@mail:~/Desktop/nodejs_dotenv/app$ docker push shefeekar/myapp:latest

the image push into the dockerhub

EC2 server provisioning with terraform attach parameter store

```
provider "aws" {
region = var.aws_region
}
```

```
##############################
```

```
SECURITY GROUP
```

```
##############################
resource "aws_security_group" "sg" {
name   = "${var.project_name}-sg"
vpc_id = var.vpc_id
```

```
ingress {
from_port   = 3000
to_port     = 3000
protocol    = "tcp"
cidr_blocks = ["0.0.0.0/0"]
}
```

```
egress {
from_port   = 0
to_port     = 0
protocol    = "-1"
cidr_blocks = ["0.0.0.0/0"]
}
}
```

```
##############################
```

IAM ROLE

```
##############################
resource "aws_iam_role" "ec2_role" {
name = "${var.project_name}-ec2-role"

assume_role_policy = jsonencode({
Version = "2012-10-17"
Statement = [{
Effect = "Allow"
Principal = {
Service = "ec2.amazonaws.com"
}
Action = "sts:AssumeRole"
}]
})
}

##############################
```

SSM PARAMETER

```
##############################
resource "aws_ssm_parameter" "my_secret" {
name  = "/${var.project_name}/${var.environment}/MY_SECRET_NUMBER"
type  = "SecureString"
value = var.MY_SECRET_NUMBER
}
```

```
##############################
```

IAM POLICY

```
##############################
resource "aws_iam_policy" "ssm_policy" {
name = "${var.project_name}-ssm-policy"

policy = jsonencode({
Version = "2012-10-17"
Statement = [{
Effect = "Allow"
Action = ["ssm:GetParameter"]
Resource = aws_ssm_parameter.my_secret.arn
}]
})
}

##############################
```

ATTACH POLICY TO ROLE

```
##############################
resource "aws_iam_role_policy_attachment" "attach_policy" {
role       = aws_iam_role.ec2_role.name
policy_arn = aws_iam_policy.ssm_policy.arn
}

resource "aws_iam_role_policy_attachment" "ssm_managed_core" {
role       = aws_iam_role.ec2_role.name
```

```
policy_arn = "arn:aws:iam::aws:policy/AmazonSSMManagedInstanceCore"
}
```

```
##############################
```

INSTANCE PROFILE

```
##############################
resource "aws_iam_instance_profile" "ec2_profile" {
name = "${var.project_name}-instance-profile"
role = aws_iam_role.ec2_role.name
}
```

```
##############################
```

EC2 INSTANCE

```
##############################
resource "aws_instance" "node_app" {
ami           = "ami-0c42696027a8ede58" # Amazon Linux 2 (us-east-1)
instance_type = var.instance_type
subnet_id     = var.subnet_id
key_name      = var.key_name
associate_public_ip_address = true
vpc_security_group_ids = [aws_security_group.sg.id]
iam_instance_profile  = aws_iam_instance_profile.ec2_profile.name
```

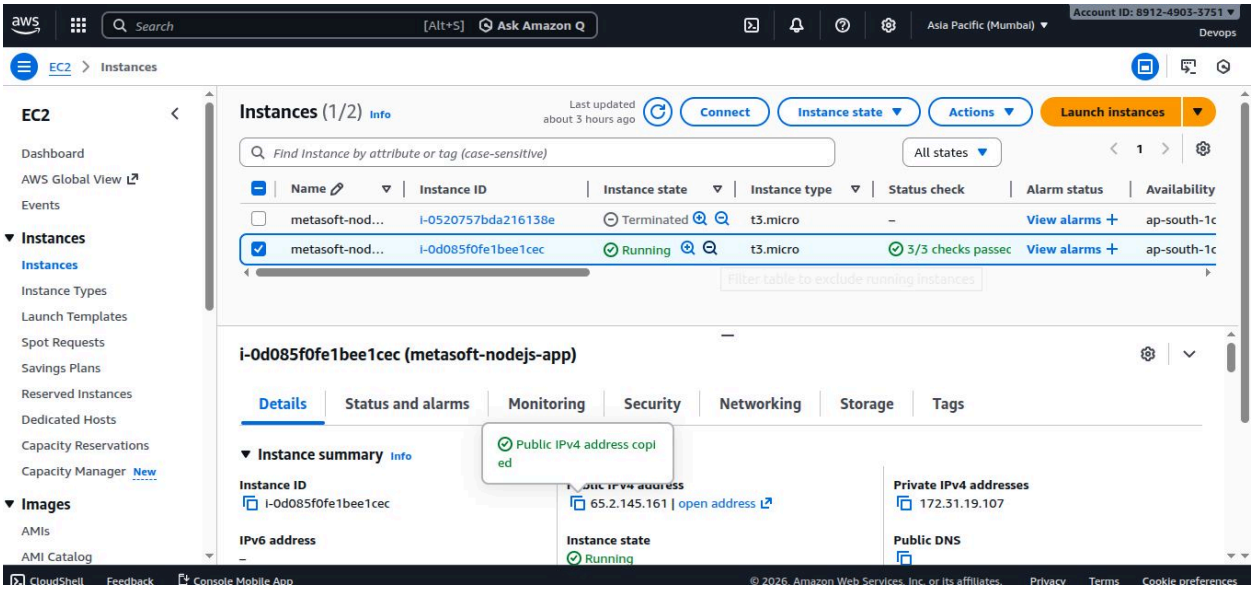
```
user_data = <<EOF
#!/bin/bash
yum update -y
```

```
cd /tmp
yum install -y https://s3.amazonaws.com/ec2-downloads-windows/SSMAgent/latest/linux_amd64/amazon-ssm-agent.rpm
systemctl enable amazon-ssm-agent
systemctl start amazon-ssm-agent
```

```
yum install -y aws-cli
yum install -y docker
systemctl start docker
systemctl enable docker
```

```
aws ssm get-parameter \
--name "/${var.project_name}/${var.environment}/MY_SECRET_NUMBER" \
--with-decryption \
--region ${var.aws_region} \
--query "Parameter.Value" \
--output text > /tmp/secret
```

```
docker run \
--name my-app \
-e MY_SECRET_NUMBER="$(cat /tmp/secret)" \
-p 3000:3000 \
shefeekar/myapp:latest
EOF
tags = {
Name = "${var.project_name}-nodejs-app"
}
}
```



Terraform Infrastructure Documentation – EC2-Based Node.js Application with Secure Secrets

Overview

This Terraform configuration provisions a **single EC2-based application stack on AWS** to run a **containerized Node.js application**.

The infrastructure is designed to demonstrate:

- Secure secret management using **AWS Systems Manager Parameter Store**
- Least-privilege access via **custom IAM role and policy**
- Application deployment using **Docker on EC2**
- Direct public access to the application over HTTP

The architecture intentionally avoids complexity such as load balancers or autoscaling and is suitable for **development, demos, and DevOps machine tests** rather than large-scale production systems.

Architecture Diagram Description

The infrastructure can be visualized as the following flow:

1. Terraform authenticates to AWS using the configured provider and region.
2. A Security Group is created inside an existing VPC.
3. An IAM role is created and configured for EC2 with:
    - Permission to read a specific SSM SecureString parameter
    - AWS-managed Systems Manager access
4. A SecureString parameter is stored in AWS Systems Manager Parameter Store.
5. An EC2 instance is launched inside an existing subnet with:
    - Public IP address
    - Attached Security Group
    - Attached IAM Instance Profile
6. During instance boot:
    - Docker and required tools are installed
    - The secret is fetched securely from SSM

    - A Docker container is started with the secret injected as an environment variable
7. The Node.js application becomes accessible on port **3000** via the EC2 public IP.

Component Breakdown

AWS Provider

The AWS provider is configured using a variable-driven region.

Purpose:

- Allows the same infrastructure to be deployed in different AWS regions without code changes.
- Assumes credentials are already available via environment variables, shared config, or IAM role.

Security Group

A dedicated Security Group controls network access to the EC2 instance.

Ingress:

- TCP port 3000 open to the internet (`0.0.0.0/0`) for application access.

Egress:

- All outbound traffic allowed.

Design decision:

- Direct public exposure simplifies access and testing.
- No SSH ingress rule is defined, implying administrative access is expected via AWS Systems Manager.

IAM Role

A custom IAM role is created specifically for the EC2 instance.

Responsibilities:

- Allow EC2 to assume the role
- Enable secure access to AWS services without static credentials

This role forms the foundation for runtime security and secret access.

SSM Parameter (SecureString)

A secret value is stored in AWS Systems Manager Parameter Store as a **SecureString**.

Characteristics:

- Encrypted at rest by AWS
- Scoped to project name and environment
- Retrieved only at runtime by the EC2 instance

Important implication:

- The secret is not hardcoded in the application image or exposed in Terraform outputs.

IAM Policy

A custom IAM policy is created to grant **minimum required permissions**.

Permissions granted:

- Read access (`ssm:GetParameter`) to **only one specific parameter**

Security posture:

- Follows the principle of least privilege
- Prevents access to unrelated SSM parameters

IAM Policy Attachments

Two policies are attached to the EC2 role:

1. Custom SSM read policy (for the application secret)
2. AWS-managed `AmazonSSMManagedInstanceCore` policy

Result:

- EC2 instance can be managed via AWS Systems Manager
- No SSH access is required for administration

IAM Instance Profile

An instance profile binds the IAM role to the EC2 instance.

Purpose:

- Enables AWS services running on EC2 to assume the role transparently
- Required for EC2 to access SSM and other AWS APIs

EC2 Instance

A single EC2 instance hosts the Node.js application.

Key attributes:

- Amazon Linux 2 AMI (region-specific)
- Public IP enabled
- Launched into an existing subnet
- Associated with the custom Security Group
- Uses the IAM Instance Profile

The EC2 instance is the **only compute resource** in this architecture.

Instance Bootstrap (User Data)

During the first boot, the EC2 instance executes a startup script that:

1. Updates system packages
2. Installs and starts the SSM Agent
3. Installs AWS CLI and Docker
4. Retrieves the encrypted secret from Parameter Store
5. Starts a Docker container with:

- Port 3000 exposed
- Secret injected as an environment variable
- Image pulled from a container registry

Notable behavior:

- The secret exists only in memory and a temporary file at runtime
- Docker container is started imperatively, not managed by a process supervisor

---

Resource & Data Flow

1. Terraform initializes the AWS provider.
2. Security Group is created inside the specified VPC.
3. IAM role, policy, and instance profile are created.
4. Secure parameter is stored in SSM.
5. EC2 instance is launched with:
   - Network access
   - IAM permissions
   - Bootstrap instructions
6. On boot:
   - EC2 authenticates to AWS using its IAM role
   - Retrieves the encrypted secret
   - Runs the Node.js application container
7. End users access the application via the EC2 public IP on port 3000.

Dependencies & Ordering

Terraform implicitly enforces the following order:

1. IAM role and policy creation
2. SSM parameter creation

3. Instance profile creation
4. Security Group creation
5. EC2 instance provisioning

Runtime dependencies (Docker, SSM agent, AWS CLI) are handled during instance boot, not by Terraform dependency graph.

Security & Networking Considerations

- Application port is publicly exposed without a load balancer.
- No inbound SSH access is defined.
- EC2 management is expected via AWS Systems Manager.
- Secrets are encrypted at rest and accessed using IAM-based authentication.
- IAM permissions are tightly scoped to a single SSM parameter.

Security trade-off:

- Simplicity is favored over defense-in-depth.

Scalability, Availability, and Resilience

This architecture is **not highly available**.

- Single EC2 instance
- No autoscaling
- No health checks
- No redundancy

Any instance failure results in application downtime until manual recovery.

Assumptions & Constraints

The configuration assumes:

- The VPC and subnet already exist

- The subnet provides internet access
- The Docker image exists and is publicly accessible
- AWS credentials are configured outside Terraform
- The application listens on port 3000

These elements are **intentionally externalized**.

Summary

This Terraform configuration demonstrates a **clear, secure, and interview-ready infrastructure design** that highlights:

- Secure secret handling with SSM
- IAM best practices
- Practical EC2-based container deployment
- Clean separation of concerns

create a dockerpush workflow using github action

name: Build and Push Docker Image to Docker Hub

on:
push:
branches:
- master

jobs:
docker:
runs-on: ubuntu-latest

```
steps:
  - name: Checkout source
    uses: actions/checkout@v4

  - name: Login to Docker Hub
    uses: docker/login-action@v3
    with:
      username: ${{ secrets.DOCKERHUB_USERNAME }}
      password: ${{ secrets.DOCKERHUB_TOKEN }}
```

```
- name: Build and push image
  uses: docker/build-push-action@v5
  with:
    context: ./app
    push: true
    tags: |
      shefeekar/myapp:latest
      shefeekar/myapp:${{ github.sha }}
```

generate git hub action secret and store in to the github secrets