

AWS ECS – API GATEWAY TASK

The architecture entails creating a VPC with two public subnets and two private subnets. These are attached to private and public route tables, respectively, along with NAT and internet gateways. RDS DB is deployed privately within the private subnet. ECS is deployed within the private subnet for security reasons. AWS ALB is then deployed, followed by a REST API Gateway with API key authentication and request limits set. Client requests are authenticated securely through API Gateway before being forwarded to ALB, ECS, and then to the DB. That's the flow.

To deploy the infrastructure, we've created Terraform scripts, along with our custom root modules. We call these root modules within child modules. Additionally, we're creating various components such as ECR, ECS, ALB, RDS, VPC, and REST API Gateway using Terraform for security purposes. To enhance security, we've set up two public and private subnets, along with Internet and NAT gateways. All configurations have been completed accordingly.

We've also created S3 buckets and implemented backend configuration to store the state file.

You can also check the resources on the AWS console.

TERRAFORM

modules	Add files via upload	1 hour ago
README.md	Add files via upload	1 hour ago
alb.tf	Add files via upload	1 hour ago
api-gateway.tf	Update api-gateway.tf	1 hour ago
backend.tf	Add files via upload	1 hour ago
ecs_cluster.tf	Add files via upload	1 hour ago
ecs_container.tf	Add files via upload	1 hour ago
graph.png	Add files via upload	1 hour ago
provider.tf	Add files via upload	1 hour ago
rds.tf	Add files via upload	1 hour ago
task-def.json	Add files via upload	1 hour ago
terraform.tfvars	Add files via upload	1 hour ago
variable.tf	Add files via upload	1 hour ago
vpc.tf	Add files via upload	1 hour ago

In our GitHub Action pipeline, we've set up a process to deploy Terraform resources on AWS. Within this pipeline, we install Terraform, initialize it, format the Terraform code, validate it, and finally apply the Terraform configuration. To maintain security, we securely store the access key and secret key in GitHub secrets, which are then utilized during the Terraform apply command. Below are the steps of our pipeline.

GITHUB ACTION CODE

```
name: Deploy Terraform

on:
  push:
    branches:
      - master

jobs:
  terraform:
    runs-on: ubuntu-latest

    steps:
      - name: Checkout code
        uses: actions/checkout@v2
```

```

- name: Configure AWS credentials
  uses: aws-actions/configure-aws-credentials@v1
  with:
    aws-access-key-id: ${ secrets.AWS_ACCESS_KEY_ID }
    aws-secret-access-key: ${ secrets.AWS_SECRET_ACCESS_KEY }
    aws-region: ap-south-1

- name: Install Terraform
  run: |
    wget https://releases.hashicorp.com/terraform/1.0.2/terraform_1.0.2_linux_amd64.zip
    unzip terraform_1.0.2_linux_amd64.zip
    sudo mv terraform /usr/local/bin/

- name: Initialize Terraform
  run: |
    terraform init

- name: Format Terraform configuration
  run: |
    terraform fmt

- name: Validate Terraform configuration
  run: |
    terraform validate

- name: Apply Terraform changes
  run: |
    terraform apply -auto-approve -var aws_access_key=${ secrets.AWS_ACCESS_KEY_ID } -var aws_secret_key=${ secrets.AWS_SECRET_ACCESS_KEY }

```

PIPELINE EXECUTION

terraform succeeded 1 hour ago in 8m 59s			<input type="text" value="Search logs"/>	
>	✔ Set up job		1s	
>	✔ Checkout code		1s	
>	✔ Configure AWS credentials		1s	
>	✔ Install Terraform		1s	
>	✔ Initialize Terraform		7s	
>	✔ Format Terraform configuration		0s	
>	✔ Validate Terraform configuration		3s	
>	✔ Apply Terraform changes		8m 43s	
>	✔ Post Configure AWS credentials		0s	
>	✔ Post Checkout code		0s	
>	✔ Complete job		0s	

After deploying the resources, we've also deployed a small Node.js application that connects to the database and inserts some sample data. We've dockerized this application and deployed it on ECS.

DOCKERFILE

```
# Use an official Node.js runtime as a parent image
FROM node:14

# Set the working directory in the container
WORKDIR /usr/src/app

# Copy package.json and package-lock.json to the working
directory
COPY package*.json ./

# Install dependencies
RUN npm install

# Copy the rest of the application files
COPY . .

# Expose the port that the app runs on
EXPOSE 5000

# Define the command to run your application
CMD [ "node", "server.js" ]
```

On the GitHub Action side, to deploy the backend application, we've used the following steps: building the Docker image, pushing it to ECR, updating the task definition with the new image tag, and deploying the task definition. Below are the steps and code of the pipeline.

GITHUB-ACTION-BACKEND-PIPELINE-CODE

```
name: Deploy to Amazon ECS

on:
  push:
    branches: ["master"]

env:
  AWS_REGION: ap-south-1 # set this to your preferred AWS
region, e.g. us-west-1
  ECR_REPOSITORY: demo-node-application-test # set this to
your Amazon ECR repository name
  ECS_SERVICE: demo-node-application-test # set this to your
Amazon ECS service name
  ECS_CLUSTER: demo-ecs-cluster-test # set this to your Amazon
ECS cluster name
  ECS_TASK_DEFINITION: demo-node-application-test # set this
to the path to your Amazon ECS task definition
# file, e.g. .aws/task-definition.json
  CONTAINER_NAME: demo-node-application-test # set this to the
name of the container in the
# containerDefinitions section of your task definition

permissions:
  contents: read

jobs:
  deploy:
    name: Deploy
    runs-on: ubuntu-latest
    environment: production

    steps:
```

```

- name: Checkout
  uses: actions/checkout@v4

- name: Configure AWS credentials
  uses: aws-actions/configure-aws-credentials@v1
  with:
    aws-access-key-id: ${ secrets.AWS_ACCESS_KEY_ID }
    aws-secret-access-key: ${ secrets.AWS_SECRET_ACCESS_KEY }
    aws-region: ap-south-1

- name: Login to Amazon ECR
  id: login-ecr
  uses: aws-actions/amazon-ecr-login@v1

- name: Build, tag, and push image to Amazon ECR
  id: build-image
  env:
    ECR_REGISTRY: ${ steps.login-ecr.outputs.registry }
    IMAGE_TAG: ${ github.sha }
  run: |
    # Build a docker container and
    # push it to ECR so that it can
    # be deployed to ECS.
    docker build -t
$ECR_REGISTRY/$ECR_REPOSITORY:$IMAGE_TAG .
    docker push $ECR_REGISTRY/$ECR_REPOSITORY:$IMAGE_TAG
    echo "image=$ECR_REGISTRY/$ECR_REPOSITORY:$IMAGE_TAG"
  >> $GITHUB_OUTPUT

- name: Download task definition
  run: |
    aws ecs describe-task-definition --task-definition
demo-node-application-test \
    --query taskDefinition > task-definition.json

- name: Fill in the new image ID in the Amazon ECS task
definition
  id: task-def
  uses: aws-actions/amazon-ecs-render-task-definition@v1
  with:
    task-definition: task-definition.json
    container-name: ${ env.CONTAINER_NAME }
    image: ${ steps.build-image.outputs.image }

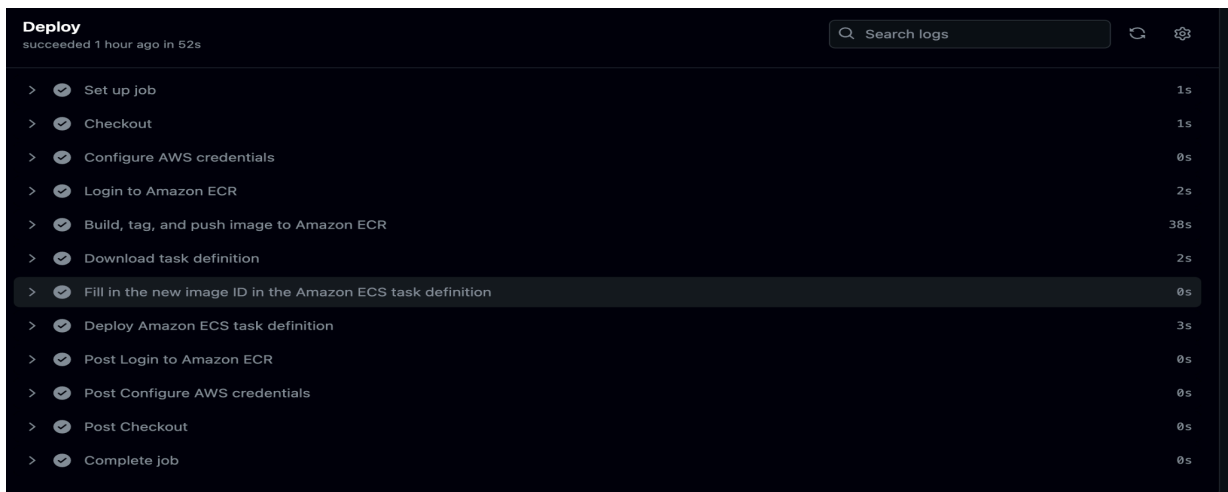
```

```

- name: Deploy Amazon ECS task definition
  uses: aws-actions/amazon-ecs-deploy-task-definition@v1
  with:
    task-definition: ${{
steps.task-def.outputs.task-definition }}
    service: ${{ env.ECS_SERVICE }}
    cluster: ${{ env.ECS_CLUSTER }}
    wait-for-service-stability: false

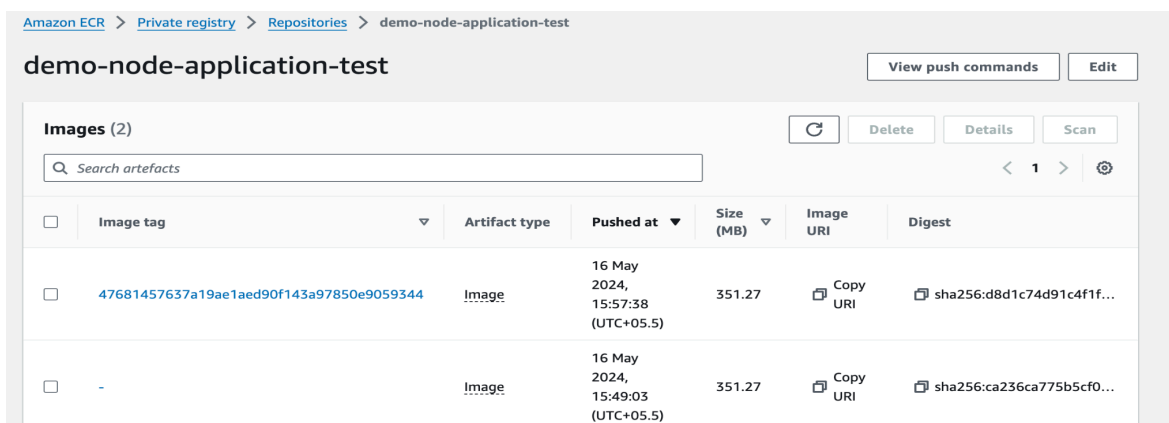
```

STEPS



You can verify the deployment of the application by logging into the AWS console and checking the ECS service running tasks.

ECR



ECS CLUSTER AND SERVICE

[Amazon Elastic Container Service](#) > [Clusters](#) > [demo-ecs-cluster-test](#) > Services

demo-ecs-cluster-test

Update cluster

Delete cluster

Cluster overview

ARN

arn:aws:ecs:ap-south-1:590183908425:cluster/demo-ecs-cluster-test

Status

Active

CloudWatch monitoring

Container Insights

Registered container instances

-

Services

Draining

-

Active

1

Tasks

Pending

-

Running

1

Services

Tasks

Infrastructure

Metrics

Scheduled tasks

Tags

Services (1) Info

Manage tags

Update

Delete service

Create

Filter services by value

Filter launch type

Any launch type

Filter service type

Any service type

< 1 >

Service name

ARN

Status

Service...

Deployments and tasks

Last...

[demo-node-application-test](#)

arn:aws:ec...

Active

REPLICA

1/1 tasks running

ECS TASK

[Amazon Elastic Container Service](#) > [Clusters](#) > [demo-ecs-cluster-test](#) > [Services](#) > [demo-node-application-test](#) > Tasks

demo-node-application-test Info

Update service

Delete service

Health and metrics

Tasks

Logs

Deployments

Events

Configuration and networking

Tags

Tasks (1/1)

Stop

Filter tasks by property or value

Filter desired status

Running

Filter launch type

Any launch type

< 1 >

Task

Last status

Desired st...

Task definition

Health s

[bf62c399c2d34daa913f1...](#)

Running

Running

[demo-node-application-test:6](#)

[Unkn](#)

And finally, you can verify the API output by accessing it through API Gateway with API key authentication

The screenshot displays a REST client interface with the following details:

- URL:** `https://nion2tckui.execute-api.ap-south-1.amazonaws.com/application/api/data/`
- Method:** `GET`
- Headers:** A table with 7 headers, including `x-api-key` with the value `C0HwrQpaB59CBKGOS0JJR95Ri7DcCRb7384xEzAS`.
- Status:** `200 OK`, `Time: 186 ms`, `Size: 858 B`
- Body:** A JSON array of 3 items, each with `id` and `name` properties.

Key	Value	Bulk Edit
<input checked="" type="checkbox"/> <code>x-api-key</code>	<code>C0HwrQpaB59CBKGOS0JJR95Ri7DcCRb7384xEzAS</code>	
Key	Value	

```
1 [
2   {
3     "id": 1,
4     "name": "Item 1"
5   },
6   {
7     "id": 2,
8     "name": "Item 2"
9   },
10  {
11    "id": 3,
12    "name": "Item 3"
13  }
14 ]
```