Flow description

Based on build, push and run methodology, this Docker + Terraform project helps to deploy a Python based Flask application into AWS Elastic Container Service (ECS) consuming images pushed on AWS Elastic Container Registry (ECR).

Briefly explained, the first step is to create the basic infrastructure for ECR. After this, a Docker image is built and analyzed. Once the image is pushed into registry, the ECS underlying infrastructure is created and endpoint connection tested.

Steps

Pre-requisites

The following requirements must be fulfilled before deploying this application:

1. An IAM credential must be generated for Terraform and Docker to create required resources and push images. Credentials should not be stored on local files. This example uses GitHub Actions secrets to pass account and secret credentials as environment variables at run-time and are never stored.
2. A Docker and Terraform ready environment. Check guides to install Terraform or install Docker (Docker Desktop is recommended) as needed.
3. An AWS S3 resource is required to store the terraform state description files as these are not being stored on local machine for credential security.

Pre-build

The following steps will create the AWS ECR resources required to store our Docker image:

1. cd into `pre-deploy` terraform project (`cd terraform/pre-deploy`) and run command `terrafom init` to initialize providers. Note: AWS providers is set to fixed version as latest version (4.56.0) is currently failing to deploy.
2. Run `terraform apply --auto-approve` to generate new infrastructure. ECR name and region can be set in the `variables.tf` file.
3. The terraform manifest will create an ECR repository and an ECR security policy to grant push access. Images are flagged as *immutable* to (comply with security](https://docs.bridgecrew.io/docs/bc_aws_general_24).
4. With this, the ECR resources should be generated and ready to store new images.

Docker build

Provided Python Flask project already comes with a valid Docker file that can be run to build the image. It is important to note that Python image selected has multiple critital and high risk vulnerabilities identified. It is recommended to increase base image version or target latest version. As this project is run from GitHub actions, source is not modified and consumed *as-is* from original repository.

Steps to build our docker image:

1. Authenticate Docker against AWS by asking for an ECR token key (most secure). `aws ecr get-login-password --region <region> | docker login --username AWS --password-stdin <aws_account_id>.dkr.ecr.<region>.amazonaws.com`. The region and AWS account id values will be required.
2. cd back to project main folder and clone python-api project with command `git clone https://github.com/mransbro/python-api`. A new folder called `python-api` will be created.
3. Once authenticated, build docker image with command `docker build --name python-api .`. This will create a new image called `python-api` following the manifest in provided `Dockerfile`.
4. Run vulnerability scan on image using free service Snyk: `docker scan --file Dockerfile --accept-license python-api`. The `--exclude-base` option can be set to bypass scanning base image. Grype can also be sued to scan images.
5. Update docker image tag to meet AWS requirements. Tag should be the same a ECR name: `docker tag python-api:latest <aws_account_id>.dkr.ecr.$region.amazonaws.com/<registry_name>:latest`
6. Publish to AWS ECR with command `docker push <aws_account_id>.dkr.ecr.<region>.amazonaws.c`
7. New image should be now available on ECR.

Post-build

Having now the AWS Elastic Container Registry ready, the next step is to crate the AWS Elastic Container Service resource and all its networking and permissions dependencies. To do so:

1. cd again to `terraform/post-build` folder and run command `terrafom init` to initialize providers. Note: AWS providers is set to fixed version as latest version (4.56.0) is currently failing to deploy.
2. Run `terraform apply --auto-approve` to generate new infrastructure. The following main resources will be created:
   - The ECS resource, its tasks and service to run container listening on `port 5000`. Including health checks based on python3 commands to test container port listening and application availability. python3 is used as curl or wget are not included on provided image.

- Two EC2 subnets, security groups, target groups and load balancer listeners on `port 80`
- An Application Load Balancer (ALB) that will help to expose the containerized application for public access

3. Once completed, the load balancer URL will be printed to allow testing. Now the deployment is completed.

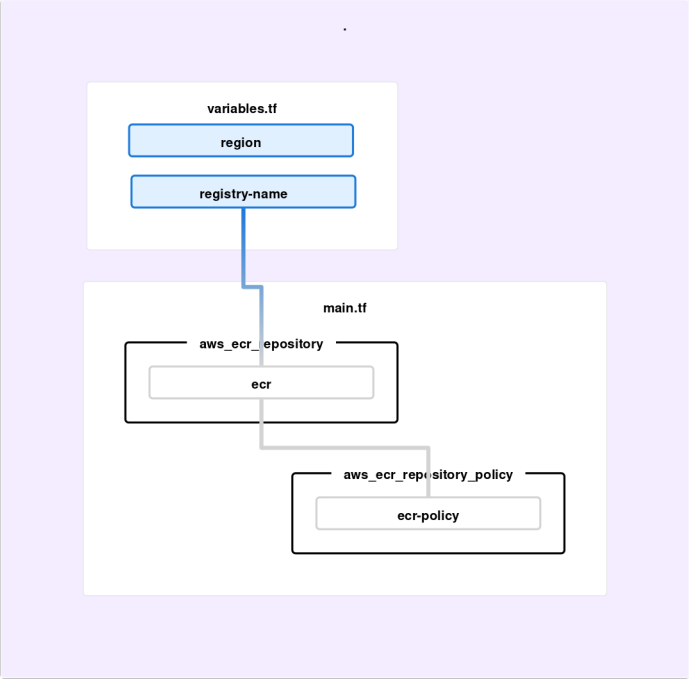
Post-deploy testing

A simple cURL test can be run from local devices to test application availability, AWS healthcheck can also be used to test application status. Consider new containers take time for initialization.

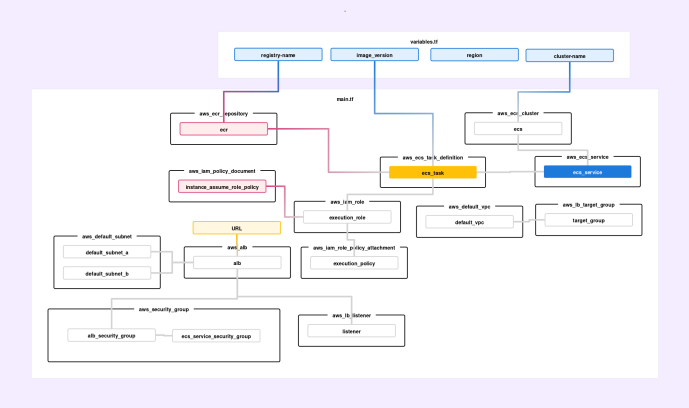
Infrastructure diagram

As infrasctructre is deployed on a two-step process to avoid usage of provisioners and maintan Terraform usage to infrastrcutre-related task only, following will be found two different graphs to ilustrate project architecture.

AWS Elastic Container Registry



AWS Elastic Container Service



Explorer

It is highly recommended to analyze deployment structure by means of a dynamic graphical tool as Rover. This can be easily achieved by running command `docker run --rm -it -p 8080:9000 -v $(pwd):/src --env-file <env_AWS_credentials_file> im2nguyen/rover -tfVar region=<region> -tfVar registry-name=<registry_name>`

Scripts

Two scripts are provided that assist to deploy and destroy the infrastructure. These can be found under the root directory, `deploy_docker_flask.sh` and `destroy_docker_flask.sh`

Scripts require variables to configure resources correcly, recommended commands are:

- Deploy

```
./deploy_docker_flask.sh -r <region> -a <account>
```

- Destroy

```
./destroy_docker_flask.sh -r <region> -a <account>
```