Patchwork DevOps project

Included with this file is a ‘deploy’ script which will set up a new Pulumi stack, build and then deploy the ‘webapp’ application to AWS.

To run this, you will need to have the following installed:

Pulumi CLI, AWS CLI, Python 3.x, Docker

You will also need to have your AWS/IAM account set up with sufficient permissions and will need to have your credentials (Access Key ID and Secret Access Key) set up in the AWS CLI (this can be done via the command ‘aws configure’)

Dockerfile

I have rewritten the supplied ‘Dockerfile’ to make use of multiple containers, the first of which is a golang container which is only used to compile the webapp. We then copy the resulting webapp binary into a new empty container.

Doing this dramatically reduces the container footprint (~550MB down to ~6MB) resulting in faster build and deployment times while also increasing stability and providing better security due to there being fewer dependencies.

Securing the application

As we now have a single binary in our docker container we have removed potential entry points therefore minimising the attack surface area and also limiting the number of failure points.

Within our Pulumi program we have only allowed access via TCP port 8080 so we are not exposing our application/server to ports or protocols it is not expecting.

However, the current application will respond to *all* requests on port 8080, meaning that it is open to denial-of-service attacks. To help mitigate this, the application should implement some form of rate-limiting for requests.

By using a load balancer in our Pulumi program we help shift load between resources to prevent overloading any one resource.

AWS IAM policies can also be used to restrict access to specific source IP address ranges, specific times of day, or other conditions.

Dealing with errors

Errors (i.e., HTTP 500) can be a sign that something has gone wrong with the configuration of the server or that the application is malfunctioning.

To be able to deal with these we need to know that these errors are occurring so we would need some monitoring solution for both the servers and the applications.

Solutions like Amazon CloudWatch or DataDog etc. can monitor the operational health of the service and if anomalous behaviour is identified can alert an engineer and can be configured to take automated actions like the rerouting of traffic to another server allowing us to minimise downtime.

Updating the application

Pulumi allows us to apply updates easily. Making changes to the ‘main.go’ file and subsequently calling ‘pulumi up’ will rebuild the app and the docker container, deploy it to ECR and reconfigure ECS to use the new container.

Further Automation

Utilising CI/CD software such as Jenkins or Github Actions could allow us to have a fully automated build, test and deployment pipeline which could be triggered by commits to the code repo or at set times of day etc.

Improvements

This project could be improved by making use of the Pulumi Automation API which would allow programmatic control of the entire stack orchestration and would enable easier debugging due to the single main entrypoint. This would save having separate shell scripts or manually issuing commands, resulting in fewer things to maintain and saving time.