Open Government Products - Best Practices

Deployment

AWS Best Practices

These are the best practices we have learnt and adopted while building infrastructure on the cloud. Most of these should apply to any cloud provider.

Do not use the root account

The root account is provided when you sign up for an new AWS account. This account has all the permissions to do all the things on your AWS account. If this account gets compromised (e.g. your AWS API secrets get pushed to GitHub), bad things will happen.

What should you do?

1. When you sign up for an AWS account, add 2FA to this root account.
2. Create a new user account and give it rights to do admin tasks

Enable MFA on all IAM account with console access

Cloud infrastructure is critical to any project and we should protect it like we do our bank accounts. MFA is easy to set up on AWS and should be mandatory. You can use the following methods for MFA:

- Software Tokens
    - Google Authenticator
    - Authy
        - Lets you backup and sync the tokens across devices. Useful when you change phones.
- Hardware Tokens
    - YubiKey

Only use your personal credentials for local development

The AWS credentials (Access key and Secret) that come with your IAM account should only be used for local development on computer. It is a bad practice to put them in TravisCI or Heroku.

Why?

Your IAM user probably has more rights than is needed by a deploy service or the actual running app. For example, the app might just need access to an S3 bucket but your IAM user will have access to entire S3 service, changing password, EC2 and databases.

What should you do?

Creat a new IAM user for each purpose and give that user just the priveleges that it needs. That means the IAM user that you are going to put on TravisCI will only need priveleges to deploy to ElasticBeanstalk. The IAM user for your app will only need access to a specific bucket in S3.

Never ever hard-code AWS credentials

Don't be lazy. From the start, just use environment variables to read credentials (or any kind of secrets) inside your code.

Why?

If you never do it, you will never have to worry about having credentials or secrets in your code ever.

How?

Here is one way of doing this when you start a project:

1. Git ignore env related files and directories

    ```
    > echo "env" >> .gitignore && echo ".env" >> .gitignore
    ```

2. Create a directory for env files. This is because your might have multiple environments

    ```
    > mkdir env
    ```

3. Create a .env file inside the directory. Name it appropriately. (e.g. heroku.env for envvars for heroku)

    ```
    > touch heroku.env
    ```

4. Add the envvars into the file

    ```
    export SOME_SECRET_FOR_SOME_PURPOSE=secret
    ```

5. Source the env file before running your app

```
> source env/heroku.env
```

You could also node module like (dotenv)[https://github.com/motdotla/dotenv] to do this.

Make use of IAM roles

If your app lives in AWS, then most likely you don't have to provide access keys to access other AWS services. Services like ElasticBeanstalk and AWS Lambda allow you to provide a IAM role during the creation process or as a config. You can then give these roles permissions to use other AWS services. EC2 instances can only be assigned an IAM role during creation so always makes sure you start instances with a role even if they don't need permissions right away.

Grant the least privilege possible

This is a standard security practice. You only need the permissions to do what you are supposed to do. If the app needs to upload an image to an S3 Bucket, then it should only have permissions to do just that. That means no permissions to view other buckets, list contents of the current bucket, remove items from the bucket or delete a bucket.

This makes it easy to audit the permissions of an app. If the app only has permissions to put an object on S3, then that's all it can ever do. Let's say you gave an app full access to S3 then it is not easy for you to say for sure what a compromised app could have done. TLDR: You reduce the attack surface in case of compromise.

Why do people not do this?

Most people grant more permissions than required because it is less tedious when developing. So if your app needs access to read a particular DynamoDB table, you just give it full access to DynamoDB. What if you need to read another table? What if you also need to do table operation like remove or add tables? What if you need to add items to the table in the future? Who wants to be going into the AWS console everytime you need additional permissions?

Why should you do this?

It is easier start with least privilege than having to remove permissions from your app while it is already running. You are likely to break your app while removing permissions from it later on in the project. So you might as well do it correctly from the start.

> "Only put off untill tomorrow what you are willing to die having left undone" - Pablo Picasso

There is a managed service for that!

Before you start configuring your own server on EC2 (IaaS), look for a managed service (PaaS, sort of) that will do it better than you.

Instead of installing node on EC2, you can use ElasticBeanstalk. Instead of installing PostgreSQL, you can use RDS or Amazon Aurora. Instead of setting up your own email server, you can use SES. You get the idea.

Why?

Managed services let you focus on developing your actual application. They make it easier to administer and scale your infrastructure.Take RDS as an example. It lets you setup automatic backups and updates of your database.

> Amazon RDS provides cost-efficient and scalable relational database capacity while automating time-consuming administration tasks such as hardware provisioning, database setup, patching and backups. It frees you to focus on your applications so you can give them the fast performance, high availability, security and compatibility they need.

Managed services are usually more expensive than running the same on EC2 but it is totaly worth the cost. Your time is more valuable to the team.

Use the right region

This is quite dumb. Make sure to check if you are in the Singapore region when you log in before you start messing with AWS. Sometimes AWS might switch your region and you will be happily setting a new RDS instance in another region only to realise you have to do it all again. We have all done this mistake before.

If it is too complicated, then you are probably doing it wrong.

Doing things on AWS should be simple and straight forward. If you find yourself doing something complicated (IP tables etc.), you are probably doing it wrong. Ask your peer for a sanity check on what you are trying to do.

Source:

- IAM Best Practices

Working with Nectar

A Collection of Non-Obvious Things About Nectar

Written by @cloudhary. Target audience: SWE unfamiliar with Nectar

What is Nectar?

Nectar is a Platform-as-a-Service product (it's actually Redhat's Openshift Enterprise) that will build and run your docker images

How do we build products for Nectar?

You need to take into account what you're able to build with so that you architect systems appropriately. You have only two things on Nectar:

- Docker containers (that you specify using a Dockerfile)
- Persistence through either volumes or a managed MySQL DB.

Do note that each container unit (CU) of Nectar gives you 1Mbps of bandwidth. It can be a bottleneck for data intensive projects.

Development

‼Containers are run as random user id that is part of root group‼
This is an important departure from what your usual workflow might have been. It is not the same as running as root. Make sure you simulate this in your dev and staging environments by doing the following:

- Run your Docker containers with the user flag, i.e, docker run –user 12345678:0
- Providing appropriate permissions to necessary directories at build time by doing RUN chmod -R g+rwx <directory of choice>. But be mindful - use the aforementioned command with caution as some important directories will not respect the modification. For example, you cannot modify or change permissions of the /etc/hosts file, or anything in the entire /etc and /var directory

Building

This is the only phase in the Nectar build process that has internet access. All dependencies should be imported and installed during this phase.

Building takes time, particularly the scanning and transferring stage. Be patient. If you’re eager to improve the situation, do things that will reduce the size of your image. Use a small base image, use fewer packages, don’t import unnecessary dependencies like nodemon, minimize layers, etc.

Do not make assumptions when building and deploying for Nectar. Where possible, use the “verbose” option (–verbose, -v etc.) to be sure of what is going on

Deploying

The easiest step of them all. Just click deploy.

Have at least two replicas running so that the underlying Kubernetes can evict any one of them without any impact to your service availability.

After clicking deploy, even if Nectar shows that the deploy was a success, be sure to check the logs to figure it out by your own.

Avoid using the health check endpoint unless you have time to figure it out and monitoring is critical for your application. It is sometimes either a liveness check or a readiness check, and if you set it up wrongly, your code may either never deploy or it may cause a healthy service to continually restart unnecessarily. They already have a monitoring service that will ping your endpoint every 5 mins and send an email to you if it fails. Just rely on that.

You need to use a forward proxy and request for credentials to communicate with any external services. To connect to your other services or the DB tier, use the environment variables

Debugging

The Nectar engineers are friendly and helpful. They are keen to see us succeed and are willing to accomodate most requests that are within their paygrade to solve. Raise a service request so that they are able to allocate resources to attend to you. Make friends with them so that you can ask for clarifications. Join the #nectar-warriors channel so that you can ask other OGP engineers who have struggled with

Once you've deployed the application, logs don't show up immediately. They take about 10 seconds to show up. Patiently wait for them and refresh the logs page.

The terminal is quite useful when debugging. You will have an easier time debugging if you install essential tools need during the build time, especially if you use base images. For example, let's say you want to check if certain routes in your app are working, you could use curl to call it from the terminal. If you didn't have it installed in the docker image, you would have to go through the entire build process again before you can continue troubleshooting. That could take about 30 mins. So make sure you install all of the tools you need for debugging. Oh, do remember that since you don't have root privileges, you are unable to run ping and traceroute. Check the linux man page to figure out if you need to be root. Usually, you're going to want to have curl installed and you'll need to make do with just that.

Some things that don't yet exist

A couple of things that you might find yourself craving that don't yet exist: Travis CI, Serverless functions, MongoDB, docker-compose, multi-stage builds, SSH into container from command line, interactive terminal (file editing software like vi don't work, for example), Auto scaling, Terraform / Infra as Code