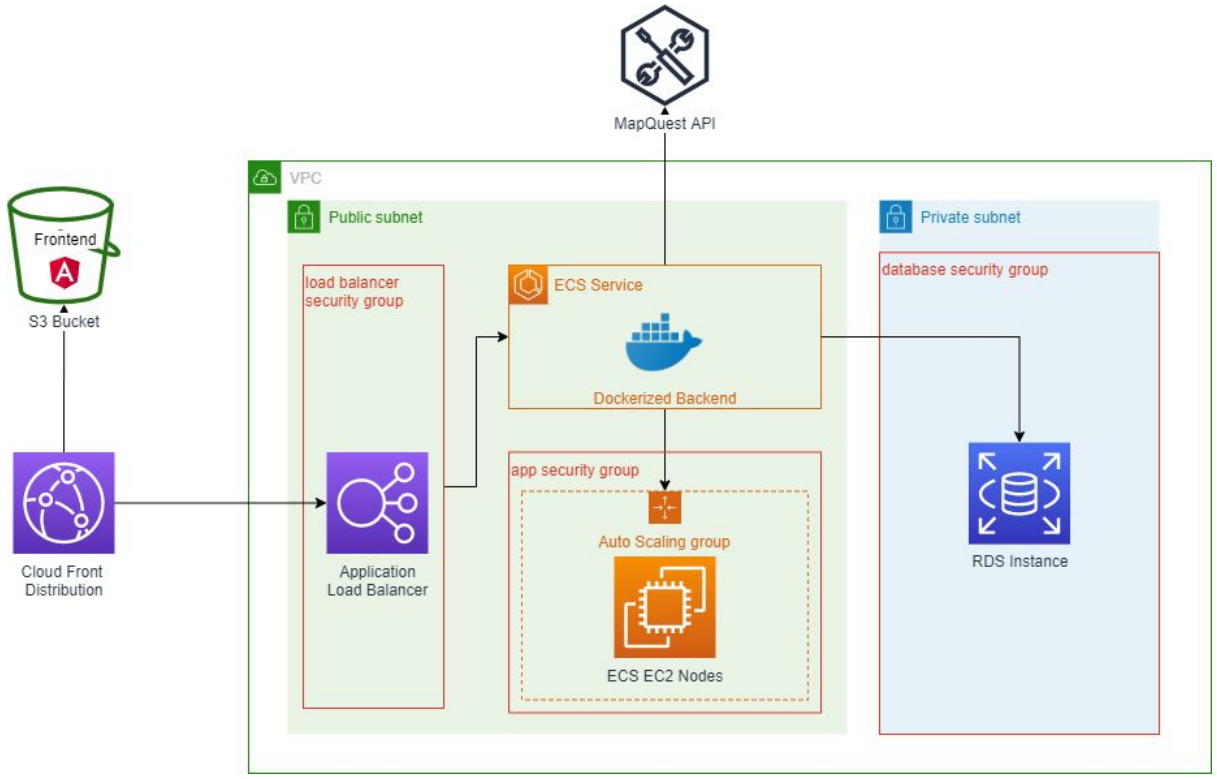
Cloud Application Architecture - Lab 4

Aim of the Laboratory

1. Switching to PaaS Database
2. Understanding Docker
3. Dockerizing our App
4. Finding and Deploying to a Suitable AWS Service

The New Architecture



Intro

Moving back to our backend application, in this lab, we will focus on reducing the stuff that we manage. Until now, we had to manage the ec2 instance where our application is running and the ec2 instance where we host our database. This means we constantly have to monitor the health of the instance and manually upgrade our application and RDBMS.

In the first part, we will focus on moving our database to a managed service – RDS (Relational Database Service).

Then, in the second part, we will focus on making our application simpler to deploy and run using Docker and migrating it to the fitting AWS service.

RDS

RDS is a managed database service. We tell AWS the hardware specs and database engine that we need (and some other info – mostly security and availability) and it will run and manage the database for us. This means that we no longer have to worry about keeping it online, applying updates and backups.

RDS supports several database engines such as MySQL, MariaDB (which is an open-source fork of MySQL), PostgreSQL, and others. We should use the same engine as we used in previous labs.

Walkthrough

1. Go with the “standard create” mode (as opposed to “easy create”) so you can see what options are available.
2. Switch between templates and look at the rest of the settings. We will use the **free tier** template that is more than enough for our needs.
3. Give it a name and set the password to **postgres**
4. For DB instance size, **db.t2.micro** should be set automatically (set it otherwise, you might have to enable *Include previous generation classes*).
5. For storage, 20GB of general purpose SSD should suffice (no autoscaling needed)
6. For the connectivity section, make sure that you configure it the same as the DB instance created in lab 2.
    - A private subnet
    - The database security group
7. Under additional configuration, set the **initial database name** to **postgres**. This is important because otherwise, RDS wouldn’t create the database for us. Also, **disable automatic backups**.
8. It will take a few minutes until the RDS instance is ready.

Deploy the App Tier

Deploy the CloudFormation stack using the following template:
https://caa-lab-templates.s3-eu-west-1.amazonaws.com/lab4.json

Update the Frontend App

Re-build and re-upload the angular application in order for it to communicate with the new backend. Follow the steps from the previous lab to provide the load balancer DNS name and upload the app to S3 (delete the previous files).

Docker

Docker is a container engine. A container is an isolated environment where we can run programs. From the application's perspective, a container is pretty much a VM. From our perspective, a container is an environment containing and running our application together with all its dependencies. A container is based on an image (if containers are objects, images would be classes).

Docker makes similar promises to what java does: you can run any application (built for Docker) on any hardware/VM that can run the engine.

You can always find out more from the Docker documentation.

Walkthrough

We will use our app's EC2 instance to get a better understanding of Docker and to "dockerize" our application. Another option would be to install the Docker engine locally, but that can be a bit painful, especially on Windows (it requires Hyper-V).

Install Docker

Look through Lab 1 to find how to install docker. After running the commands, you might also need to run *newgrp docker* to avoid having to re-log.

Create your first container

Docker images are hosted on the Docker Hub repository (there are other container registries). Run a new container instance based on the *hello-world* image (follow the instructions at https://hub.docker.com/_/hello-world)

List containers on your machine

Use *docker ps* to display the list of containers (Hint: you will need the –a option to display stopped instances)

Create your own container image

Creating a new docker image means telling docker the following:

- The starting image (e.g. Ubuntu)
- What other programs to install
- What files to copy into the container (e.g. our application's .jar file)
- How to run it

This is all achieved using the *Dockerfile*.
Create your own image using a custom Dockerfile. Here is a sample Dockerfile that uses a Node.JS script to print "Hello World":

```
## use node js base image
FROM node:alpine
```

create JS source file
RUN echo "console.log('Hello from Node.JS')" > index.js
Tell Docker what command to run when the container is created
CMD node index.js

Build a docker image based on your Dockerfile using the *docker build* command:

docker build –t my_first_image .

(note the final ".", which is the path to the application sources – in this case, the current directory)

Run the image using *docker run my_first_image --name first*

Clean Up

Remove the hello-world container and the images:

docker rm first

docker rmi my_first_image

Dockerizing/Containerizing our app

Dockerizing our application means creating a JVM-based container image and copying the application binaries (the jar file) into the container image

Use https://spring.io/guides/gs/spring-boot-docker/ as a guide.

For the ENTRYPOINT, you can get the exact command by looking for the process where the app is running (*ps aux | grep java*). (The tutorial uses exec form, we will use shell form - i.e. the one from the running process)

Push the image to the AWS Container Registry (ECR)

1. Create a registry from the console
2. Authenticate on ECR with

docker login -u AWS -p $(aws ecr get-login-password) https://$(aws sts get-caller-identity --query 'Account' --output text).dkr.ecr.eu-west-1.amazonaws.com

3. Follow this guide:
https://docs.aws.amazon.com/AmazonECR/latest/userguide/docker-push-ecr-image.html

Deploying to AWS

AWS offers a wide selection of services that you can use to run a container. Check the 3 options below.

Service	Documentation
AWS ECS	ECS is Amazon's proprietary container orchestration service.

	https://aws.amazon.com/ecs/
AWS EKS	With EKS you get your own Kubernetes cluster, managed by AWS. https://aws.amazon.com/eks/
Fargate	Fargate is AWS's serverless offering for containers, meaning you don't have to manually allocate EC2 resources since it does it for you
Elastic Beanstalk	EB is a higher-order service that orchestrates ECS and ELB (and other) AWS resources https://aws.amazon.com/elasticbeanstalk/

Read the documentation and identify 1 scenario in which each of the above services is best suited for the job.

Deploy your application using AWS ECS

Create an ECS cluster

An ECS cluster uses EC2 instances as underlying infrastructure on which to deploy container instances. Create a new ECS cluster following the tutorial at https://docs.aws.amazon.com/AmazonECS/latest/developerguide/create_cluster.html.

Use the following settings:

- Cluster Template: EC2 Linux + Networking
- Provisioning Model: On-Demand
- Instance Type: t2.micro
- Number of instances: 1
- AMI: Linux 2 AMI
- VPC & subnet: select the existing VPC and subnet (same as the previous app instance)
- For the container instance role, select the role created by the CloudFormation stack (the name starts with the name of the stack).

After completing the setup go to the EC2 service. Note that there isn't any new EC2 instance running (yet). However, there is a new auto-scaling group which will start up the EC2 instance on-demand, I.e. as soon as the first task is started

Create a Task Definition

The task definition tells AWS ECS how it should start your container.

https://docs.aws.amazon.com/AmazonECS/latest/developerguide/create-task-definition.html

Create a new task definition, set a name, and proceed to add a container definition in the task:

- Specify the container image name (copy it from ECR)
- Set a memory limit (512 MB)

Run the Task / Create an ECS Service

Creating the task definition does not yet start any container instance. For that, you can either manually run the task, or create an ECS service, which will ensure that a predefined number of container instances is available at all times. The service also exposes your container through a load balancer.

To create a service navigate to your ECS cluster and select *create service*:

- Launch Type: EC2
- Task Definition: <<select your task definition>>
- Number of tasks: 1
- Load balancer type: application load balancer
- Load balancer name: <<select your existing load balancer>>
- Choose the existing listener port and create a new target group. Further instructions are available here: https://docs.aws.amazon.com/AmazonECS/latest/developerguide/service-create-loadbalancer-rolling.html

Test the App

If you navigate to the Load Balancer's Target Group, you should have 2 instances - one from the CF stack and one from the ECS cluster. Feel free to deregister the one from the beginning.

Cleanup

Delete the following:

- ECS cluster
- ECR image
- RDS instance (no final snapshot)
- CloudFormation stack