Cloud Application Architecture- Lab 1

Contents

Aim of the Laboratory

- Set the goal for this semester
- Access your AWS account
- Familiarize yourself with the AWS console, regions and services
- Create an EC2 instance
- Run the "legacy" online shop application in a "lift and shift" manner on EC2

The Goal

Migrate and adapt a traditional application to AWS cloud.

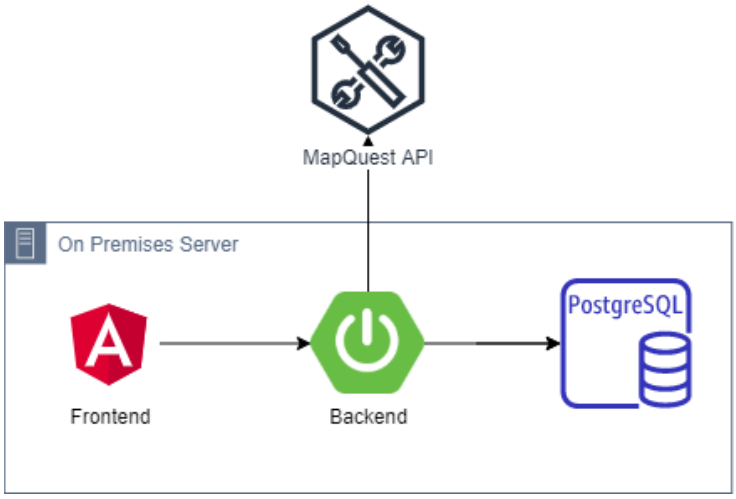


Figure 1 Initial Architecture

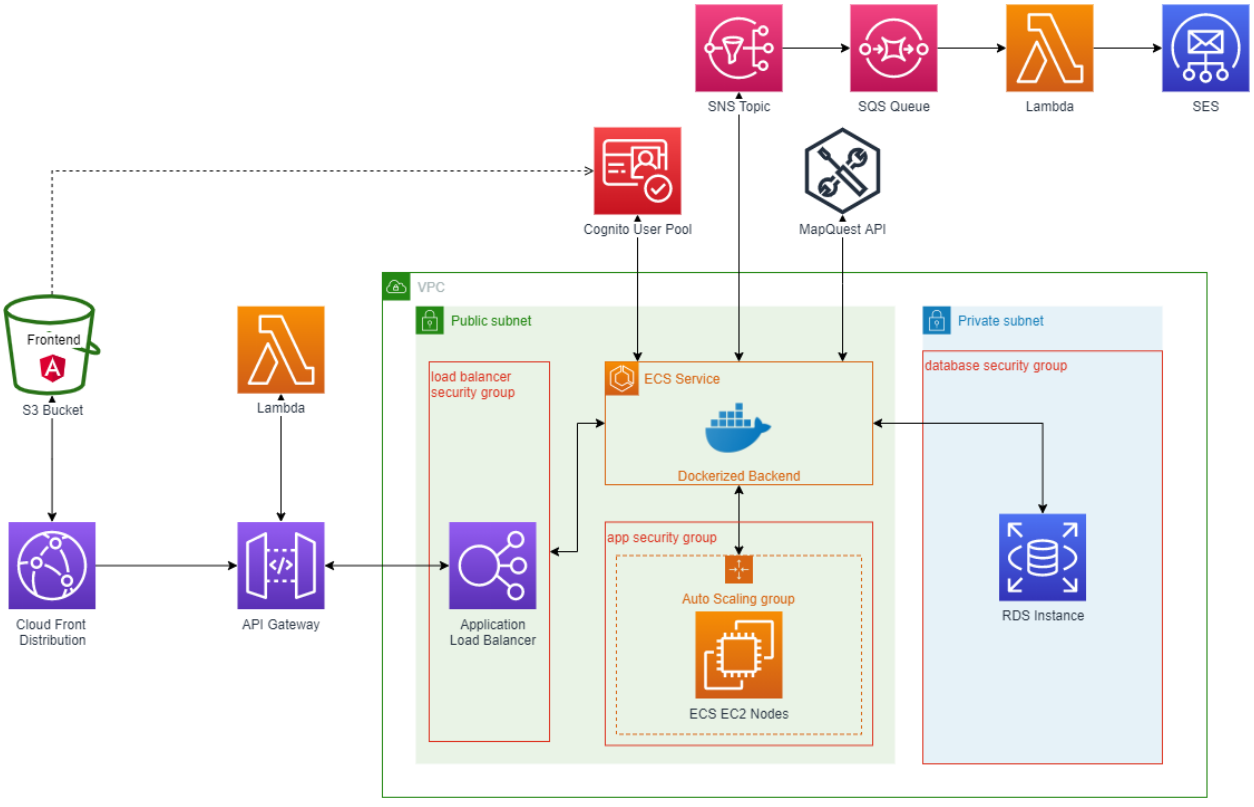


Figure 2 Final Architecture

Introduction into AWS - important concepts and terminology

AWS Intro

AWS is one of the many cloud providers available today. This wasn't always the case. Amazon launched it in 2006 as a way to increase their revenue by renting their computing resources - they had a lot of spare capacity left after scaling their resources for holiday sales. It took several years for any competitor to respond which translated to an overwhelming market share owned by AWS even today and to more mature services.

It currently offers over 200 services for various use cases (hosting, AI, ML, IoT, media encoding and streaming etc.). Some of them are very specific to a certain use case while others are more general and most likely are the building blocks of the rest. We will focus on the latter.

AWS Regions, AZs

"Amazon cloud computing resources are hosted in multiple locations world-wide. These locations are composed of AWS Regions, Availability Zones, and Local Zones. Each *AWS Region* is a separate geographic area. Each AWS Region has multiple, isolated locations known as *Availability Zones*."

An overview of all AWS regions is available at https://www.infrastructure.aws/

The AWS Console

The **AWS Console** is a management interface that acts as a central entry point towards all AWS services. In addition to the console, AWS cloud resources can also be managed through the AWS **CLI** or programmatically by using the AWS **REST API/SDK**.

"Online Shop" Application Overview

In this semester you will be dealing with an existing Online Shop application. Throughout the laboratories, you will transform the architecture and move from a "legacy" application to a fully cloud-native architecture leveraging multiple managed services of AWS.

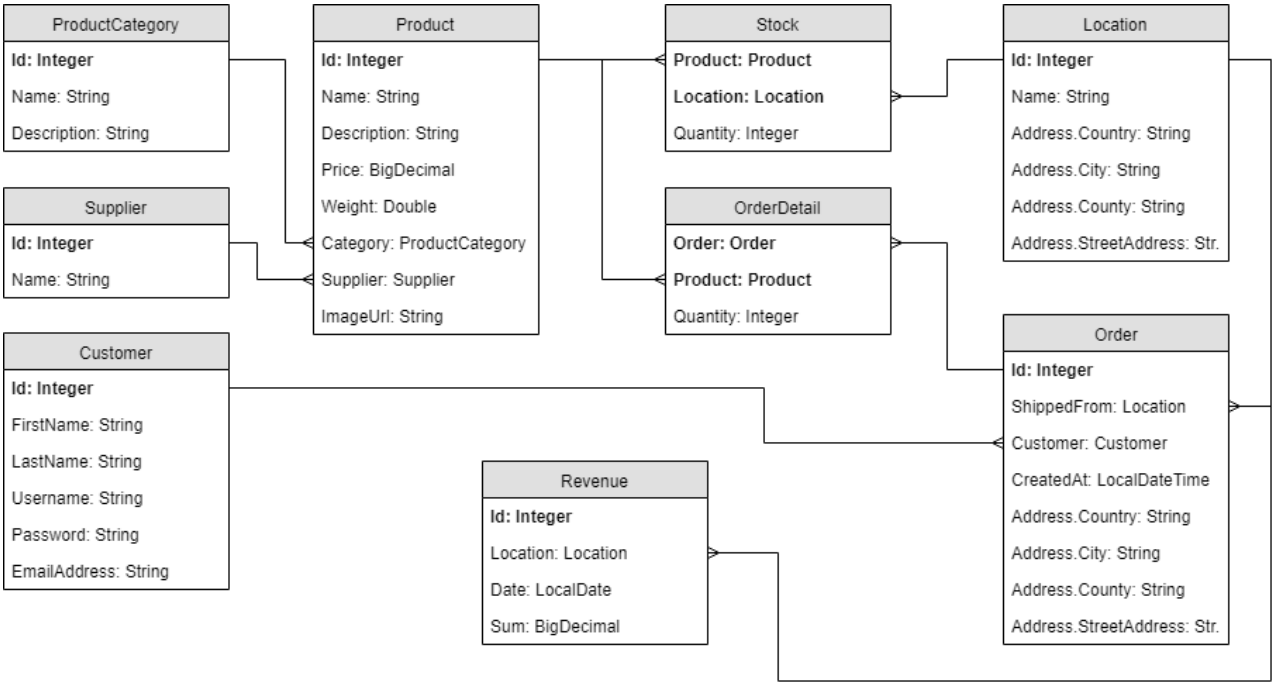
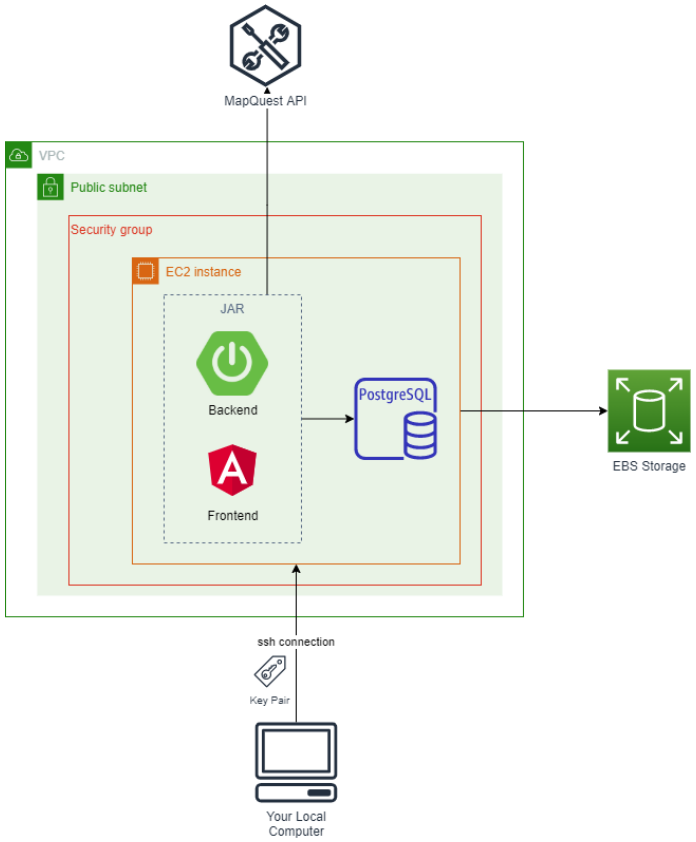


Figure 3 Data Model

Source code for the application is available here.

At the end of this lab, our application will look like this:



Lab Walkthrough

Access your AWS account

AWS Console URL: https://console.aws.amazon.com/console/home

Throughout this semester you will work in your own AWS account. For credentials, ask the lab assistant.

Navigate through the AWS services

The ***Services*** menu entry on the upper left side of the console lists all available AWS services. Have a look at the Compute, Storage and Database sections and see if you can answer the following questions:

- What Database service does AWS offer for NoSQL databases?
- What is the S3 service?
- What service can be used to build applications using a FaaS (function as a service) paradigm?

If you cannot answer all questions, no worries, we will have a look at them in the later part of the semester.

Select a region

A dropdown in the upper-right side of the console lists all available regions. Select *eu-west-1* (Europe Ireland).

> **Note**
>
> Regions serve multiple purposes. They are fundamentally different/independent clouds. They even receive new services at different rates (Ireland is the first one to receive new stuff in Europe). They are highly relevant when your application is addressing the global market (e.g. Netflix) and/or when you cannot accept any kind of downtime (in extremely rare cases, an entire region might go down).

Create an EC2 instance backed by EBS storage

We will now create our first AWS virtual machine.

Launch Instance

Navigate to the *EC2* service and, in the *instances* section, select “Launch Instance”.

Select AMI

> **Note**
>
> An Amazon Machine Image (AMI) provides the information required to launch an instance**, including the operating system and additional software**. You must specify an AMI when you launch an instance. You can launch multiple instances from a single AMI when you need multiple instances with the same configuration.

Select **Amazon Linux 2** for the AMI (64-bit x86 or Arm). This an AWS-optimized Linux flavor that can be used as part of the AWS free tier.

Instance Type

> **Note**
>
> Amazon EC2 provides a wide selection of instance types optimized to fit different use cases. Instance types comprise varying combinations of CPU, memory, storage, and networking capacity and give you the flexibility to choose the appropriate mix of resources for your applications.

Select a general purpose **t2.micro** instance type. This provides a machine with 1 vCPU and 1 GB memory.

Configure Instance Details

On this section you define further details such as the virtual network (VPC) and subnet to which the VM is allocated. Take a quick look at the explanations of some of the options.

Leave the default values and scroll down to the User Data section. Here you can define a script that will be executed automatically. We will use it to install some prerequisites on the machine (such as the java runtime environment etc.).

User Data:

```
#!/bin/bash
sudo yum -y update

# Install Maven and set the environmental variables
wget http://mirror.olnevhost.net/pub/apache/maven/maven-3/3.6.3/binaries/apache-maven-3.6.3-bin.tar.gz
tar xvf apache-maven-3.6.3-bin.tar.gz
sudo mv apache-maven-3.6.3 /usr/local/apache-maven
echo 'export M2_HOME=/usr/local/apache-maven' >> /home/ec2-user/.bashrc
echo 'export M2=$M2_HOME/bin' >> /home/ec2-user/.bashrc
echo 'export PATH=$M2:$PATH' >> /home/ec2-user/.bashrc
sudo rm apache-maven-3.6.3-bin.tar.gz

# Install Amazon Correto (AWS's implementation of the JDK)
sudo amazon-linux-extras enable corretto8
sudo yum install -y java-1.8.0-amazon-corretto-devel

# Remove the default JDK
sudo yum remove -y java-1.7.0-openjdk

# Install git
sudo yum install -y git

# Install Docker for running Postgres
sudo amazon-linux-extras install docker
sudo service docker start
sudo systemctl enable docker
sudo usermod -a -G docker ec2-user

# Pull, configure, and run the Postgres image
```

```
docker pull postgres
mkdir ${HOME}/postgres-data/
docker run -d \
        --name dev-postgres \
        -e POSTGRES_PASSWORD=postgres \
        -v ${HOME}/postgres-data/:/var/lib/postgresql/data \
    -p 5432:5432 \
    --restart always \
    postgres
```

Could we also automate the deployment of our application in this way?

Storage

The Amazon Linux 2 image uses an attached 8GB network storage by default. No need to change anything here.

Tags

> **Note**
>
> A tag is a label that you assign to an AWS resource. Each tag consists of a key and an optional value, both of which you define. Tags enable you to categorize your AWS resources in different ways, for example, by purpose, owner, or environment.

The bigger the project/system, the more important tags get.

Add the tag **project**: **caacourse** to both the instance and the EBS volume. This tag will help us track the resources and costs generated by the lab. **We will use this tag for all resources from now on.**

Security Group

We will connect to this instance via SSH, that's why it is important to leave port 22 open for incoming connections. However, you should not be doing this in a production environment as it exposes your instance to security threats.

Also, since the application will be running on port 8080, allow incoming connections on this port as well (as a custom TCP rule.

Review and Launch

Take a quick glance at all the configuration and proceed.

Key Pair

In order to connect to the instance, you will need to create an SSH key pair. Select "Create new key pair", provide a name, download the file and keep it safe.

Launch the EC2 instance and select view instance. It might take a few minutes until it's ready – use this time to configure ssh access.

Connect to your instance via SSH

> **Note**
>
> Main cloud providers, including AWS, provide a simpler way to access our instances directly from the browser. In the case of EC2 instances, after selecting the instance, there should be the **Connect** button which will open a shell connected to the instance in a new browser tab. However, since not all providers have this option, practicing the universally accepted way (with ssh) might be useful in the future.

Prerequisites

- **Your public IP address and DNS name**

By default, the instance receives a public IP address and a DNS name (based on the IP address). This should be visible on the "Description" tab of the EC2 service. Copy it as you will need it for the ssh connection.

- **SSH Client**

You will need a **ssh client** to connect to your instance.

a) Linux

Use the command line ssh client. On Linux you also need to make sure that the private key file is not publicly viewable on your computer. Use "chmod" for this

> chmod 400 my-key-pair.pem

b) Windows

Either use putty (https://www.putty.org/) or the command line client that comes with git bash (https://gitforwindows.org/). Newer builds of Windows 10 also come with a built-in ssh client (OpenSSH) (more info here).

For using putty, take a look at the official AWS guide.

For using the command line tool:

> ssh -i /path/my-key-pair.pem ec2-user@<public_ip_address>

Confirm that you trust the authenticity of the host (type yes).

Legacy Application "lift and shift"

Now that you have connected to your own EC2 instance, it is time to run the Online Shop on the VM.

Download the release from Github and run it:

> source ~/.bashrc

> curl -L -O https://github.com/CAA-Course/app/releases/download/1.0/shop-1.0.jar

Start the application

> java -Dspring.profiles.active=with-form,local -jar shop-1.0.jar

Your application should now be accessible at the above-mentioned public IP address (don't forget about the port). Try it out!

The default credentials are:
User: **user**
Password: **storepass**

Hint: you can stop the app by pressing Ctrl+C. Another useful tip when working with Linux is how to exit vim – press Esc and type ":wq" (w = write/save, q = quit).

Bonus Task

Configure the app to run as a Linux service. In our case, this is nice to have since we will stop the instances at the end of each lab. In practice, this is mandatory to avoid downtime as much as possible.

Hint: One way is using *systemd* and *systemctl*.

Cleanup

In the cloud world you pay for what you use, so make sure you stop the EC2 instance before the end of the laboratory. If you remember to stop your instances after each lab, you get a bonus point at the lab exam.

From the AWS Console, select the instance and go to **Actions**. In the **Instance state** menu, press **Stop instance**.

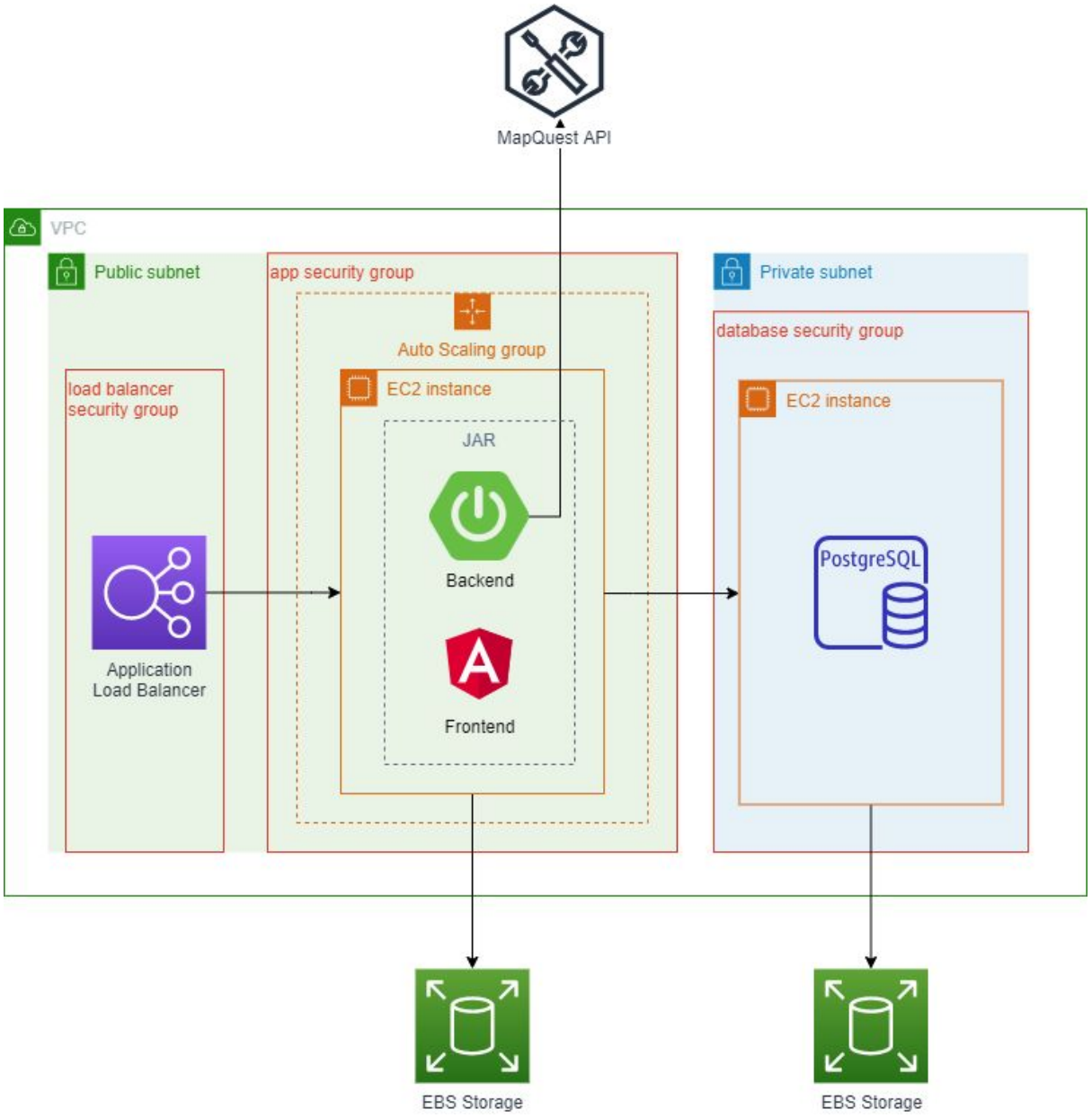
Important: When you stop an instance, the public IP address will be released. When you start the instance again, it will have a free IP address (and DNS name).

Cloud Applications Architecture - Lab 2

Aim of the lab

- Understand AWS networking concepts
- Decouple application and database layers
- Make the application layer scalable

The New Architecture



Concepts and Terms

CIDR IP Ranges

Classless Inter-Domain Routing (CIDR) is a method for allocating IP addresses and for IP routing. IP addresses are described as consisting of two groups of bits in the address: the most significant bits are the network prefix, which identifies a whole network or subnet, and the least significant set forms the host identifier, which specifies a particular interface of a host on that network. This division is used as the basis of traffic routing between IP networks and for address allocation policies.

Whereas classful network design for IPv4 sized the network prefix as one or more 8-bit groups, resulting in the blocks of Class A, B, or C addresses, under CIDR address space is allocated to Internet service providers and end-users on any address-bit boundary. In IPv6, however, the interface identifier has a fixed size of 64 bits by convention, and smaller subnets are never allocated to end-users.

IP address construction:

Class A network: Everything before the first period indicates the network, and everything after it specifies the device within that network. Using 203.0.113.112 as an example, the network is indicated by "203" and the device by "0.113.112."

Class B network: Everything before the second period indicates the network. Again using 203.0.113.112 as an example, "203.0" indicates the network and "113.112" indicates the device within that network.

Class C network: For Class C networks, everything before the third period indicates the network. Using the same example, "203.0.113" indicates the Class C network, and "112" indicates the device.
More about CIDR: https://en.wikipedia.org/wiki/Classless_Inter-Domain_Routing

VPC

Amazon Virtual Private Cloud (Amazon VPC) lets you provision a logically isolated section of the AWS Cloud where you can launch AWS resources in a virtual network that you define. You have complete control over your virtual networking environment, including selection of your own IP address range, creation of subnets, and configuration of route tables and network gateways. You can use both IPv4 and IPv6 in your VPC for secure and easy access to resources and applications.
More about VPC:
https://docs.aws.amazon.com/vpc/latest/userguide/what-is-amazon-vpc.html

Subnets

The way IP addresses are constructed makes it relatively simple for Internet routers to find the right network to route data into. However, in a Class A network (for instance), there could be millions of connected devices, and it could take some time for the data to find the right

device. Therefore, subnetting comes in handy: subnetting narrows down the IP address to usage within a range of devices.

Because an IP address is limited to indicating the network and the device address, IP addresses cannot be used to indicate which subnet an IP packet should go to. Routers within a network use something called a subnet mask to sort data into subnetworks.
More about subnets: https://www.cloudflare.com/learning/network-layer/what-is-a-subnet/

Security Groups

A security group acts as a virtual firewall for your instance to control inbound and outbound traffic. When you launch an instance in a VPC, you can assign up to five security groups to the instance. Security groups act at the instance level, not the subnet level. Therefore, each instance in a subnet in your VPC can be assigned to a different set of security groups.
More about security groups:
https://docs.aws.amazon.com/vpc/latest/userguide/VPC_SecurityGroups.html

Load Balancer

Load balancing refers to the process of distributing a set of tasks over a set of resources (computing units), with the aim of making their overall processing more efficient.
Elastic Load Balancing, the AWS service for loading balancing supports the following types of load balancers:

- **Application** - This is an actual proxy between the internet and your application. It receives a request from a client and makes another request (with the same data) to your application. It offers tons of features and it suits very well in most cases. One important tip about it is that since the ALB creates another request, but, for some reason, you need the IP address of the original client, you can look at the request header x-forwarded-for.
- **Network** - You can look at it like at a (very sophisticated) network router. While the ALB operates at layer 7 (HTTP, WebSockets) of the OSI model, the NLB handles traffic at layer 4 (TCP/UDP) thus working with packets. You lose some features of the ALB, but gain massive performance (and scalability) and the request looks like it came directly from the original client. Also, interestingly enough, it is (slightly) cheaper than an ALB (mostly because you have to configure it more).
- There is a third option, classic, but it's deprecated.

More about load balancing:
https://docs.aws.amazon.com/AmazonECS/latest/developerguide/load-balancer-types.html#clb

Auto-Scaling Groups

An Auto Scaling group contains a collection of Amazon EC2 instances that are treated as a logical grouping for the purposes of automatic scaling and management. An Auto Scaling group also enables you to use Amazon EC2 Auto Scaling features such as health check

replacements and scaling policies. Both maintaining the number of instances in an Auto Scaling group and automatic scaling are the core functionality of the Amazon EC2 Auto Scaling service.
More about auto-scaling groups:
https://docs.aws.amazon.com/autoscaling/ec2/userguide/AutoScalingGroup.html

Lab Walkthrough

1. Understand the default VPC

Navigate to the AWS VPC service and analyze the default VPC. Answer the following questions:

- What is the IPv4 CIDR range of the VPC? ([first IP] - [last IP])
- What subnets are available by default? Are they public or private?

2. Create a private subnet

A subnet is a sub-range of IP addresses within the VPC. AWS resources can be launched into a specified subnet. Use a public subnet for resources that must be connected to the internet and use a private subnet for resources that are to remain isolated from the internet. In this task, you will create a private subnet into the default AWS VPC.

- Go to the **VPC service**
- In the left navigation pane, click *Subnets*.
- Click *Create subnet* and configure:
    - *Name tag*: Private Subnet
    - *VPC*: There should be only one, the default one
    - *Availability Zone*: Select the first AZ in the list (do not choose *No Preference*)
    - *IPv4 CIDR block*: 172.31.64.0/24

Your Default VPC now has 4 subnets (3 default public and the 1 private you created).

What is the IP range of your new subnet? (first and last IP address)

3. Configure the security groups

We need a security group for the application layer allowing app traffic and, optionally, SSH, and another security group for the database layer allowing traffic from the application layer. We will start with the application layer:

- Navigate to the **EC2** service
- Go to **Security Groups** (navigation pane on the left)
- Click Create security group and configure:
    - Security group name: **App-SG**
    - Description: Allow application traffic
    - VPC: **the default VPC**

    - **Add an inbound rule** to allow traffic on the port of the app from all sources
    - Add the tag **project: caacourse**
    - **Create** then close

Now, let's create the database security group. Follow the same steps, but name it **Database-SG** and **allow traffic for Postgres only from App-SG**.

4. Spin up the instances

We are going to use CloudFormation to provision our database and application instances. For this, open the CloudFormation service and press **Create Stack** (with new resources):

- *Template is ready* option should be selected
- In the Amazon S3 URL paste the following: https://caa-lab-templates.s3-eu-west-1.amazonaws.com/lab2.yaml
- Press next
- Give your stack a name (*Lab2* maybe?)
- Fill in the parameters so CloudFormation configures the instances exactly as we need (these are part of the template you are using). Each of them should have a description.
- For ***DbImageId***, your job is to find it. Hint: It refers to an **AMI** image ID and the owner is *026709880083*.
- Press next and create the stack.

Until the instances are created and initialized, download the template from above and take a look at it.
The app should be up and running in ~2 minutes;

Next, we want to introduce horizontal scaling (i.e. increase the number of instances running the app). The main steps to achieve this are as follows:

- Create a **Launch Template**
- Create an **Auto-Scaling Group**
- Create a **Load Balancer**

5. Create the launch template

We use launch templates to tell AWS how to launch new instances when needed.
Create a new one by right-clicking on the app instance and pressing *Create template from instance*:

- Give it a meaningful name
- Check the *Auto Scaling guidance* checkbox.
- AMI and instance type should already be set
- Select the app security group
- Remove the network interface
- In the advanced details set the *Shutdown behavior* and *Stop - Hibernate behavior* to "*Don't include in launch template".*
- *Create the launch template*

6. Create the ASG

Using the launch template, we now can create an auto-scaling group that will ensure that our required number of instances is fulfilled:

- Go to Auto Scaling group (left menu)
- Press Create
- Give it a meaningful name and select the template you just created
- On the next step, for the subnet, set at all 3 default subnets
- Skip step 3
- On step 4 set the **desired capacity to 1**, **minimum capacity to 1, and maximum capacity to 2.**
- Skip the notifications, add the usual tag, and create the ASG.
- Go to the EC2 instance list. A new instance should be initializing. Add the already existing app instance to the ASG (right-click, instance settings).

7. Create the Load Balancer

Follow the official guide to create an Application Load Balancer (without HTTPS). Remember to create a new security group allowing traffic on port 80 (don't use the one for the app). The ALB will then route the traffic on the app port (8080; set on step 4).

After it's created, attach it to the ASG by following this guide.

8. Test the app

What do you notice? Can you explain?
Look through the load balancer settings. Can you identify any settings that might help?

9. Cleanup

Delete all the resources (we will start with another CloudFormation template in the next lab):

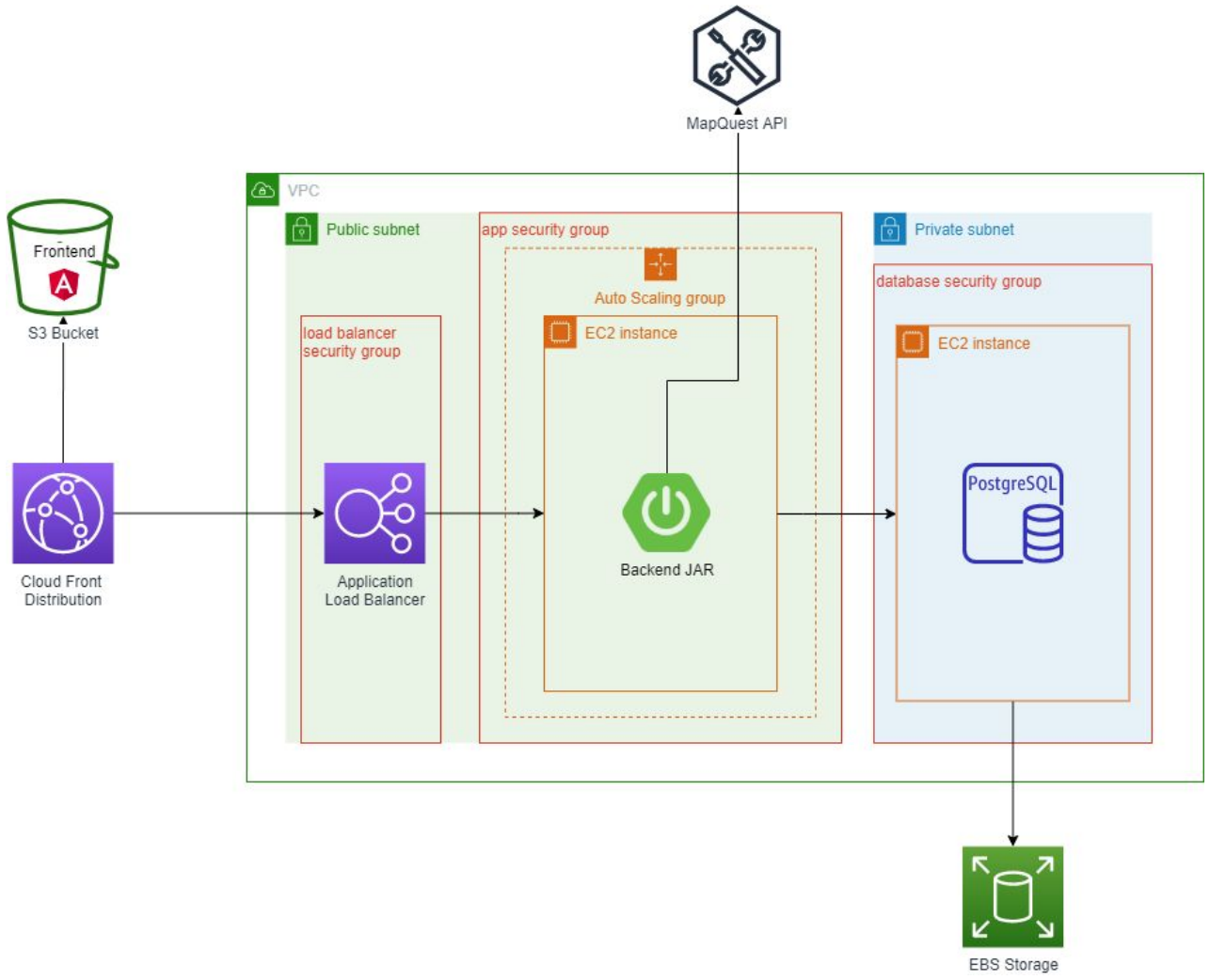
- ALB
- ASG (this should also terminate the instances that are part of it)
- CloudFormation stack (this should terminate the DB instance)
- EC2 instances if any left

Cloud Application Architecture – Lab 3

Aim of the Laboratory

- Familiarize yourself with the front-end of the application
- Connect it to the application
- Deploy it on AWS
- CDN

The New Architecture



Intro

In the previous lab, we extracted the **database** into a separate EC2 instance, worked with **networking** services, and made our **application tier highly scalable**.

Another important component of our system is the **front-end**. Right now, the application tier serves the web pages. While this works fine for many cases, it adds unnecessary load on all components of our system, which also leads to unnecessary charges (due to increased traffic and resource usage).

Since the last decade, the approach has been a bit different: let the back-end do its job (i.e. process data) and move the presentation layer (front-end) somewhere else. This approach is also encouraged by the emerging UI frameworks that we have today (react, angular, vue, polymer, svelte, and many more). Such frameworks generate static files (HTML, CSS, JS) and load the data from the back end (usually through HTTPS requests) at runtime (i.e. when the user visits our website). Since they are static files, we don't really need computing resources to host them (i.e. the host doesn't have to do any processing other than sending the files as they are to the client/requester).

In this lab, we will deploy the front-end application separately using suitable AWS services and connect it to our application (which will act just as a back-end).

Getting started with the app

Before we proceed, please make sure you have the following tools installed on your machine:

- git (check with command ***git --version*** - should be greater than 2.27.0)
- node (check with command ***node -v*** - should be greater than v12.16.1, but older versions should also be enough)

We will extract the frontend out of our application. You can find the app here:

https://github.com/CAA-Course/app (frontend directory)

Clone the repo on your computer and let's take a look at it (navigate to a suitable directory and run ***git clone <link from above>***).

Being an **angular** application, it uses **npm** to manage its dependencies (e.g. angular itself is installed as a library and a dependency of our UI app). Npm comes bundled with Node.js so you should already have it (you can check using the command ***npm -v*** - you should see something like 6.13.14). You can find more about npm here or by simply googling it (it's by far the largest package registry so you will most likely find something).

When working with such JS apps, the usual steps involve installing the dependencies (listed in *package.json*) with the command **npm install**, filling some environmental variables if necessary, and running the app with **npm start**, **npm run start**, or **npm run dev** (it really depends on the developers of the application and how they defined the script).

Consuming Our Backend

It's time to bring some actual data into our new web app. For that, we will configure it to communicate with our existing application running on EC2 behind a load balancer.

TODO:

- Find where to configure the server hostname
- Hint: these settings are usually stored in **environment** files
- Update the target to the DNS name of the load balancer

Once done, you should be able to run the app locally (npm run start) and retrieve data.

Building the frontend app

Generate the production build of the frontend app with **npm run build**.

Finding a Suitable AWS Service for Deploying It

AWS offers a huge array of services, some of them being used more than the others. Just like EC2, **Simple Storage Service (S3)** is without a doubt one of the most used services out there. What does it do? Simple. It stores data (objects). It is highly durable (your data will not get corrupted), highly available (it basically never goes down) and allows you to store virtually an infinite amount of data.

Another important aspect is that it works with **HTTP(S)**. If, for example, an EBS volume must be attached to an EC2 instance in order to access it, S3 is accessed directly through HTTP(S) without having to mount it to anything.

When we want to store files in S3, we first create a bucket (a container for our files). Each bucket has its own URL in the form of
https://<bucket_name>.s3.<region>.amazonaws.com/<file_name>

Object vs Block Storage

We are most likely used to block storage. This is what our computers use to store all our data (and OS and programs). Files are divided into blocks of data. By contrast, object storage stores files as one big chunk of data and metadata that is identified by an ID (usually the name of the file). The main consequence of this is that object storage doesn't allow us to update/edit files. If we want to update a file, we re-upload the whole file.

Bucket Access

Buckets are private by default. This means that only the owner (the AWS account) of the bucket can access it. We can give access to other AWS accounts, or, if needed, we can make it public. AWS deliberately made it slightly more complicated to make buckets public in order to avoid/reduce data leaks. Of course, if we use a bucket to host our website, we should make it public.

Static Hosting

Since S3 already serves our files through HTTPS, it takes only one small step to serve our website – telling S3 which file to serve as the index of the website (i.e. the file that is served when our users access the URL of our bucket). This usually is *index.html*.

The URL of our website will be *http://<bucket_name>.s3-website-eu-west-1.amazonaws.com/* (notice the slight difference from the usual bucket URL).

Deploying to S3

TODO:

- Create a bucket
- Make it publicly accessible
    - Disable *block public access*
    - *Add a bucket policy which allows read access for all files*
- Configure it for hosting
- Upload the web app (only the built files, contents of the **dist/online-shop** directory)
    - **Make the files publicly accessible (Read object)**
- Access it in your browser

You can always consult the official tutorial here.

One major issue of using S3 for hosting is that it doesn't support HTTPS (only plain HTTP) (this only applies to **hosting**; S3 works with HTTPS for other scenarios). The usual approach involves CloudFront, another AWS service.

CDN

A CDN (content delivery network) is used to serve files as fast as possible around the globe. It is basically a highly distributed network, thus being geographically close to as many users as possible. AWS's CDN is CloudFront and the network is made of edge locations.

We can use CloudFront to distribute **any HTTP accessible** data, including our website hosted on S3.

TODO:

- Create a web distribution for your bucket. The relevant fields are:
    - Origin Domain Name: your bucket
    - Restrict Bucket Access: select 'Yes'. Only CloudFront should be able to access your bucket.
    - Origin Access Identity: create a new identity
    - Grant Read Permissions on Bucket: we want CloudFront to update the bucket policy automatically
    - Set the **default root object** to *index.html*
    - ~~Viewer Protocol Policy: we want to redirect all plain HTTP traffic to HTTPS.~~

It will take a few minutes for your distribution to be deployed. Use this time to understand the generated bucket policy.

Once your distribution is deployed, you can access it using the domain displayed in the **general** section. You will notice that you get automatically redirected to the bucket URL. This is due to DNS propagation on the AWS side. After a few hours, this should no longer happen.

Question: How would you test CloudFront from multiple locations/continents?

Configure HTTPS

Cleanup

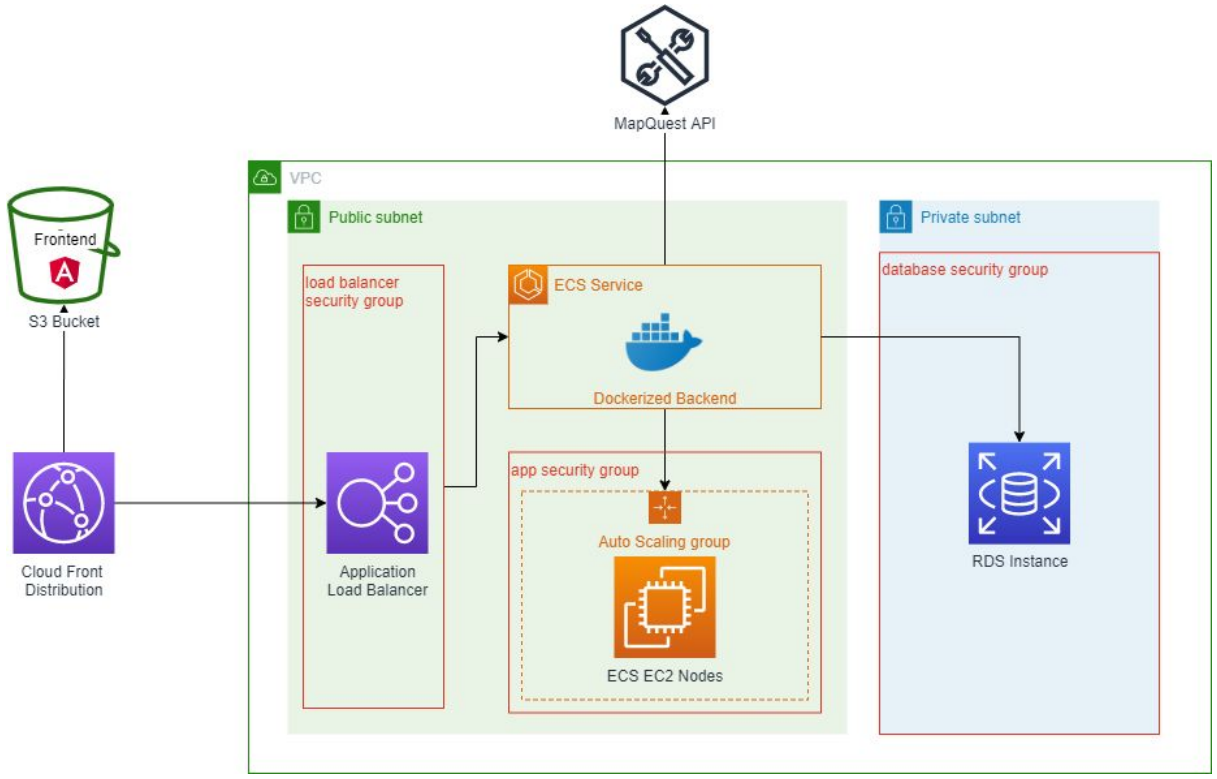
Delete the CloudFormation stack. Keep the CDN and the S3 bucket.

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

Cloud Application Architecture - Lab 5

Aim of the Laboratory

1. Complete previous labs
2. Recap of previous labs
3. IAM

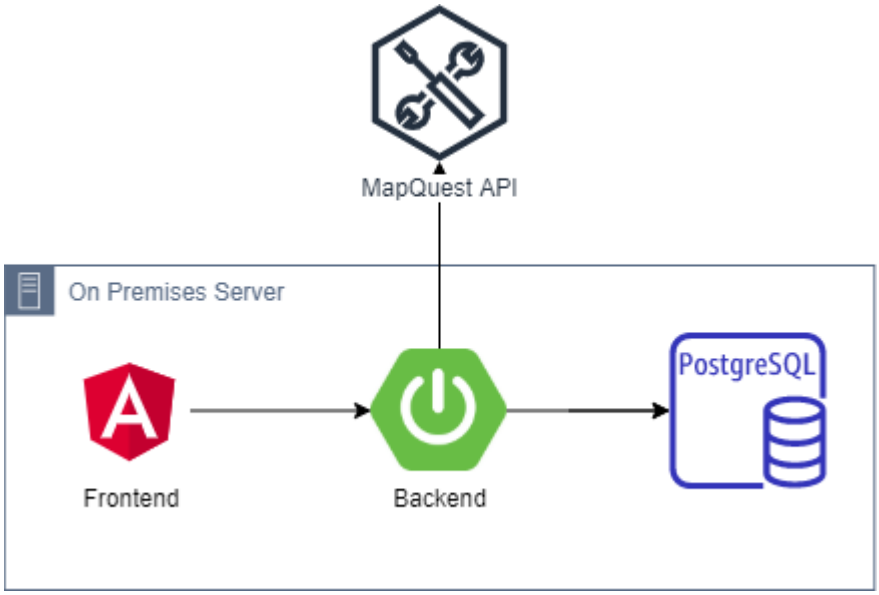
Completing Previous Labs

The first half of the lab is about getting a better understanding of the previous labs. If there are any tasks that you want to try again (or didn't manage to do them the first time) or if certain parts were not clear, now is the time to work on them.
The main points so far were:

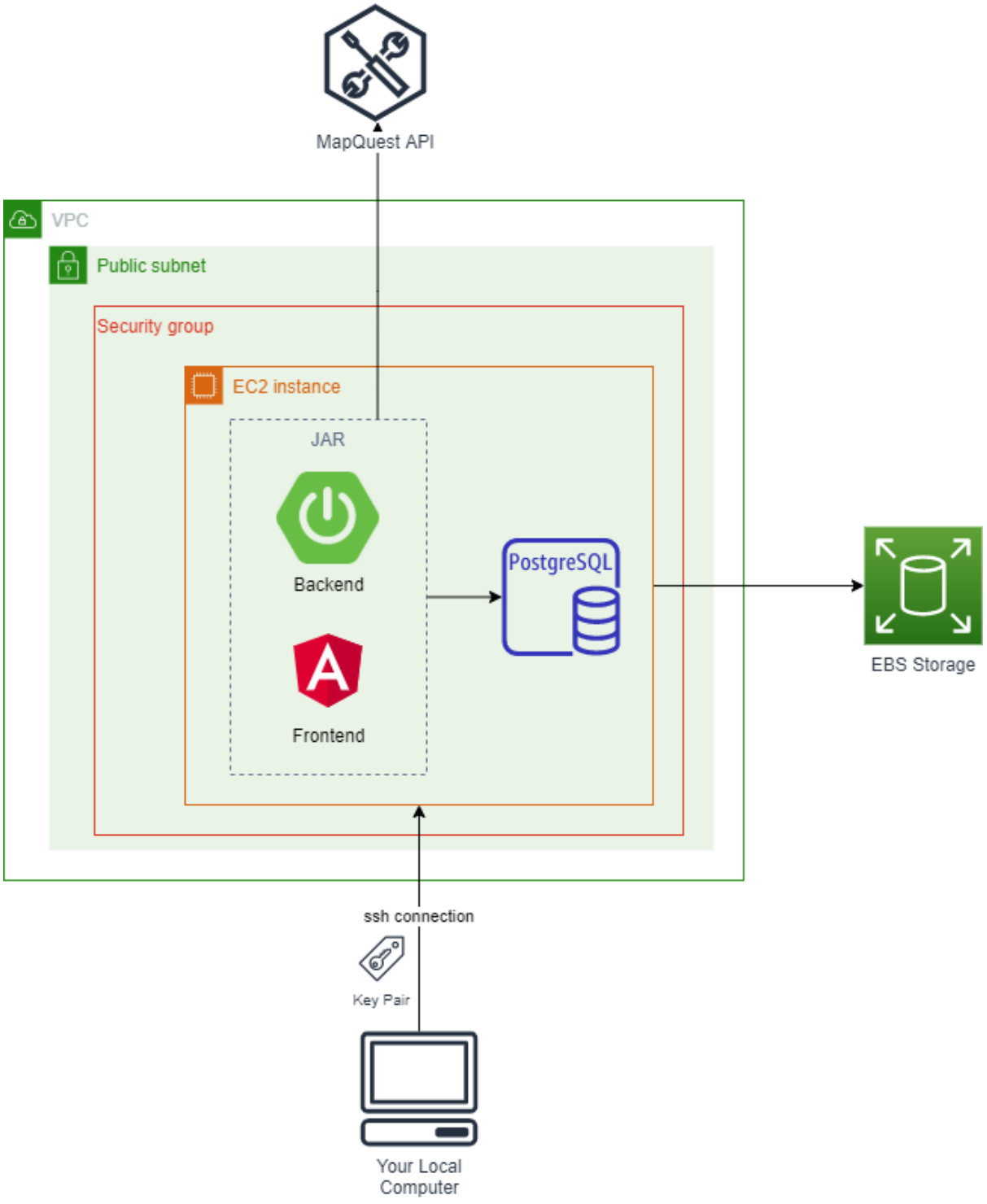
- The AWS account and how to use it (we used the web console so far)
- SSH
- EC2 instances
- VPCs, Subnets, and Security Groups
- Auto-Scaling Groups and Load Balancers
- S3 and CloudFront
- CloudFormation
- RDS and containers related services

The Road so Far

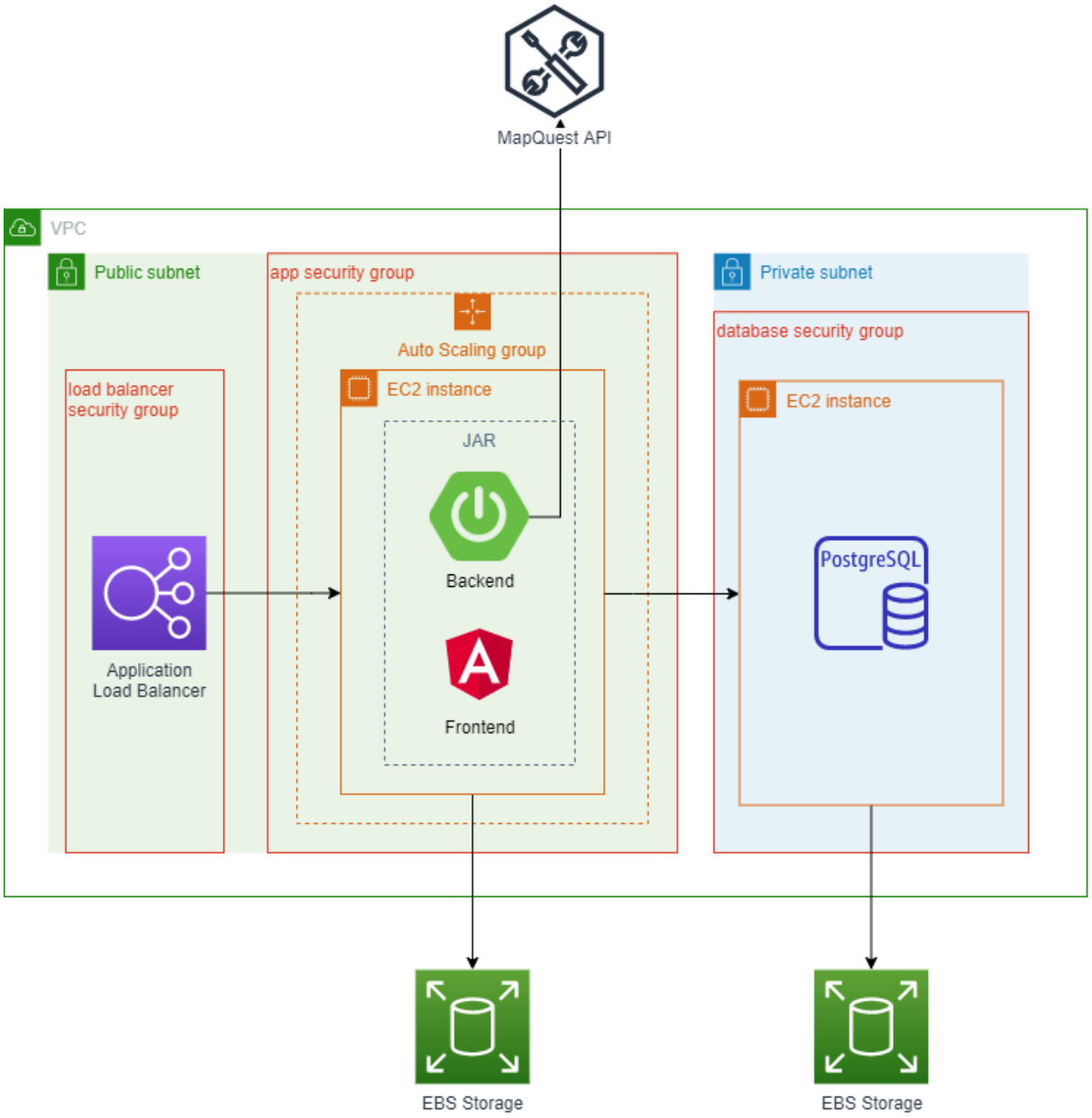
1. On-Premises



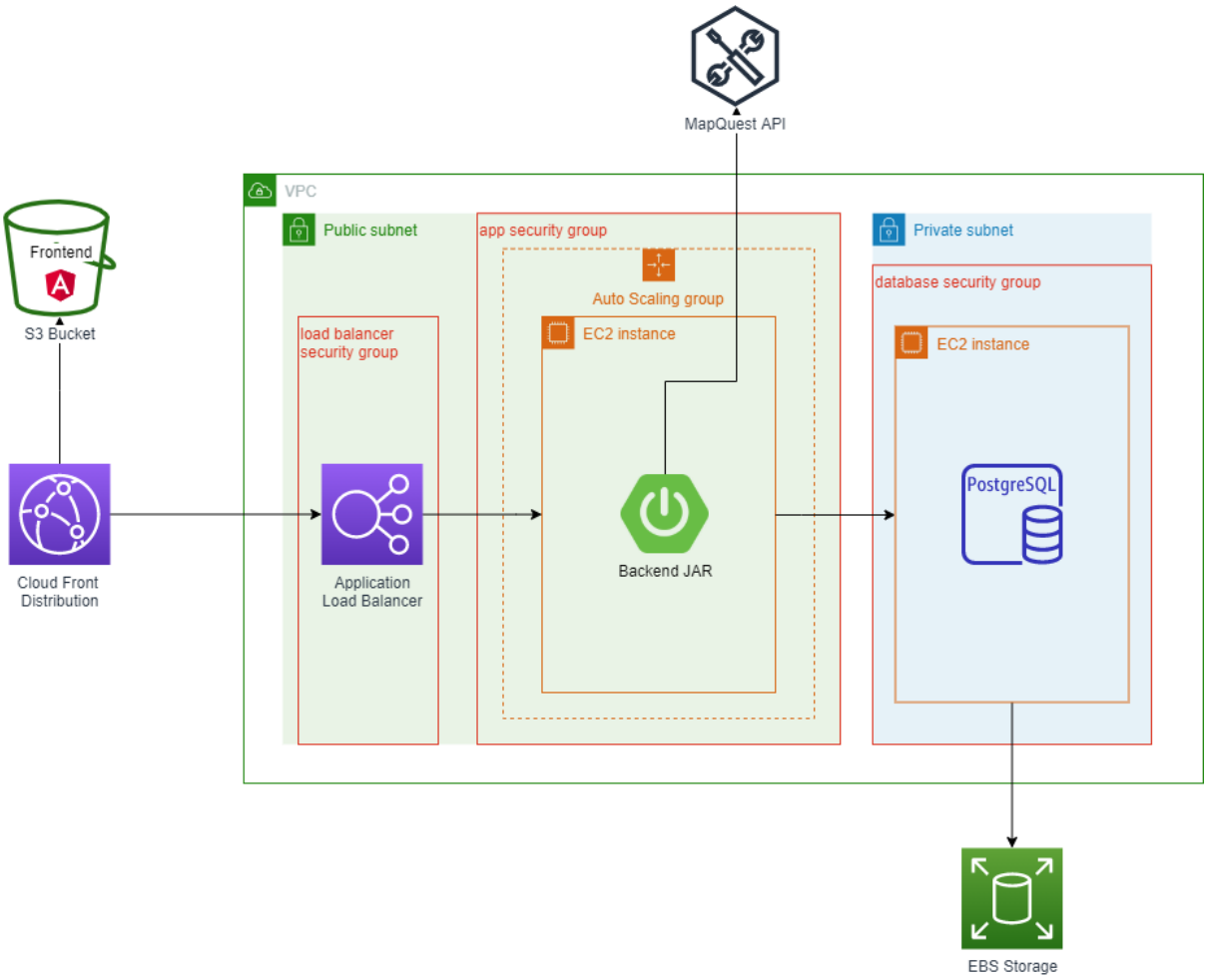
2. Lift & Shift with EC2



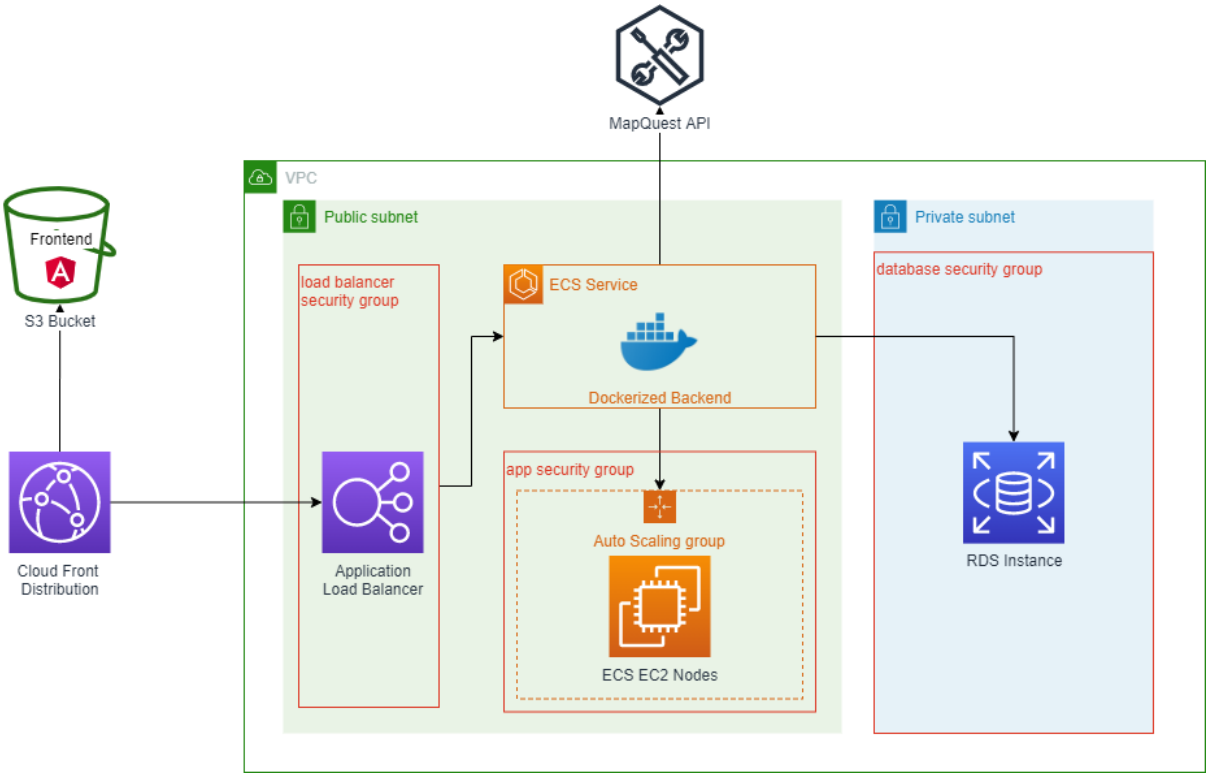
3. High Availability and Scalability



4. Frontend on Separate Hosting



5. Leveraging Managed Services and Containers



Identity & Access Management

This is the central service/place to manage everything related to access across all AWS services. This is the first service you should interact with after creating a new AWS account.
Your first account is called the root account. This is similar to Internet Explorer; you should use it only to create other 'accounts' (users). The main reason/concern is security of course - if someone gains access to your root account there is no quick way to stop him/her which could lead to considerable expenses.

Today we will assume that you are responsible for managing the users of your account. You have a new developer that joined your team and he needs **full access to EC2** and **read access to RDS**.
Your task is to create a new group called **Devs** with the right policies attached and a new user **John** for your new teammate.
Your user, *student*, might not be allowed to create IAM entities. To temporarily elevate your privileges, assume the role **IAM-Admin-Role** (the account ID is the one from the top-right menu).

Users

IAM users represent entities that interact with the AWS resources under your account. They are basically a set of credentials (username/password or access keys) that offer access to the AWS APIs/console. Each user is given a certain set of permissions adhering to the least privilege principle.
For example, in our case:

- Someone (or something) created an AWS account
- Authenticated using the root credentials
- Created an admin user (all permissions)
- Switched to that user
- Created your user and assigned the required permissions to interact with the VMs, DBs, logs, and other services covered during this training (by default a user cannot do anything)
- Sent you the username and password

Question: Do users expire? How about passwords?

Groups

Groups are what you probably expect to be - groups of users. The main benefit is that groups allow us to maintain permissions of multiple users (the members of a group). In the previous example about how your user was created, the administrator did not specify each permission individually (even though it is possible); he/she just

assigned your user to the 'Students' group from where your user inherited the permissions.
There are no pre-defined groups. In our case, the admin created the 'Students' group that you are part of.

Policies

At a high level, policies are documents (JSON) describing what an entity can or cannot do. There are 2 main types of policies (these are the most relevant for developers):

- Identity-based: they specify what the entity having it attached can do.
- Resource-based: they are attached to resources (most commonly to S3 buckets) and describe who/what has access to it and to what extent - the who/what is called the principal.

There are other policy types like policies to define permission boundaries which are like policies for policies.

Of course, you can always check the AWS documentation here. Besides some intimidating blocks of texts and diagrams, it has pretty good example policies to cover certain scenarios.

Roles

Roles do not represent an entity, but they can be assumed by an entity. You can use roles to provide temporary access to AWS resources of your account to your applications or to external users (from another AWS account or outside of AWS - e.g. identified by Google). When you create a role, you define who/what can assume it and what permissions it provides.

This is the preferred way to provide permissions to your applications. If we wouldn't have roles, we would have to provide credentials to our applications (i.e. store the username and password on the VM where the app is running).

Cleanup

Delete the following:

- ECS cluster
- ECR image
- RDS instance (no final snapshot)
- CloudFormation stack
- Any manually created EC2 instances and/or load balancers
- The IAM Users, Groups, Policies, and Roles

Cloud Application Architecture - Lab 6

Aim of the Laboratory

1. Delegating user management (Cognito)
2. Setting up the API Gateway
3. Working with Queues and Lambda

Intro

Our app is working great. It does what it's supposed to do, it is secure and it scales decently. In many cases, the current state might be good enough especially if your aim is not to reach Facebook/Amazon-like scale. However, if you do, there are a few things that can be done to facilitate that.

Right now, the app is a **monolith**. There is one artifact/service providing and handling all the functionality. The bigger the monolith gets, the harder it will be to add features since the whole application has to be built, tested, and deployed. Moreover, if there is a bug making the app unusable, the whole system becomes unusable.

The natural approach and way forward is to break the application into separate services (a.k.a. microservices). This might be a good medium to long term goal, but in the short term, we can choose to apply this approach only for new features and/or to extract certain features/domains that are clearly divided (e.g. user management).

User Management

Our spring application does not have to handle users. Instead, we can use dedicated services that can help us manage users and offer the most common operations such as password reset, email verification, and so on.

Cognito

Cognito is the AWS service dedicated to user management. It has 2 components:

- **User pools**
- Identity pools

For our purpose, we will leverage **user pools**, identity pools enabling us to grant access to the AWS account/infrastructure itself (think of it as social sign-in for your AWS account). We will create a user pool that will also enable us to adopt a new authentication/authorization scheme based on OAuth.

Follow the next steps:

1. Go to Cognito, choose Manage User Pools
2. Create a new user pool named caacourse. Choose **Review defaults**
3. Edit the attributes (first section) as follows:
    a. Change to **Email address or phone number** instead of Username - **Allow email addresses**
    b. Take a look at what attributes you can collect.
    c. Can you think of any custom attributes that might make sense to collect? (during user registration)
4. You might want to relax the password policy to make it easier
5. On step MFA and verifications, change the recovery to **Email only**
6. Take a look at the Message customizations page
7. On step App clients, add a new app client as follows
    a. Give it any name you wish
    b. **Uncheck** the **Generate client secret**
    c. Check Enable username password-based authentication (ALLOW_USER_PASSWORD_AUTH)
    d. Create it
8. Take a look at the triggers page.
    a. What would be one use for the **Pre sign-up** trigger?
9. Review and create the user pool.
10. Copy the user pool ID.
11. Go to the App client settings and enable Cognito User Pool as the **Identity Provider**.

Feel free to take a look at the AWS Docs for User Pools (there are a lot of features we don't cover in the lab).

Users

The usual approach involves having a client/web application that has the Cognito SDK (actually part of the amplify framework) installed and calls its methods to sign-in/up users, generate tokens, and so on. However, we will create a user manually and use the **AWS CLI** to validate it (change its password) and retrieve the **idToken** which will enable us to call our application.

- Create a new user under **Users and Groups**
    - Use an email address as the username
    - Don't set a phone number (also uncheck the verified box)
    - Set the email address and make sure it is verified automatically (the checkbox below is set)
- Either install the AWS CLI and configure it or launch a new EC2 instance (Amazon Linux 2 based instances have the AWS CLI already installed) and enable it to call Cognito by creating a role. If you choose to launch a new EC2 instance, first create an IAM role as follows:
    - Go to IAM, Roles
    - Create a new role for EC2

    - Search for the Cognito Power User policy and attach it.
    - Name your role something suggestive, maybe ec2-cognito-admin-role.
    - Assign the newly created role to the new instance.
    - SSH into the instance (using the Connect button or an SSH client)
- Initiate the auth flow with the following command

aws cognito-idp initiate-auth --auth-flow USER_PASSWORD_AUTH --auth-parameters USERNAME=<your_cognito_user_email>,PASSWORD=<your_cognito_user_password> --client-id <your_cognito_app_client_id>

- Respond to the password challenge with the following command (Cognito requires the users to change their passwords if they were created by the admin):

aws cognito-idp respond-to-auth-challenge --client-id <your_cognito_app_client_id> --challenge-name NEW_PASSWORD_REQUIRED --session "<session_id_from_previous_response>" --challenge-responses USERNAME=<your_cognito_user_email>,NEW_PASSWORD=<your_cognito_user_new_password>

- It's easier to assemble the command in a text editor. The session id might contain some new line characters which have to be removed (you can also search for online tools)
- The response should contain an **IdToken**, **RefreshToken** and **AccessToken**. We need the **IdToken**.
- If the token expires, you can retrieve a new one using the first command (with *initiate-auth*).

Deploy the Application

We will leverage CloudFormation again to set up our infrastructure (RDS, ECS, ALB). Create a new stack using the following template:

https://caa-lab-templates.s3-eu-west-1.amazonaws.com/lab6.json

You only have to provide the URL based on which the application will perform the token validation (it will use the URL to fetch various information about the identity provider - your user pool). (Leave the other parameter with the default value).

While the deployment takes place, make sure you have an API client (e.g. Postman) available so we can test our app.

After the deployment is complete, test the API as follows:

- Get the ALB DNS name and append */api/products* to it.
- Set the authorization to bearer and provide the idToken retrieved using the AWS CLI
- You should get a few products as a response

Extend the App

To make our lives easier, we will deploy an additional service in front of our application - API Gateway. Follow these steps:

- Go to API Gateway in the AWS console
- **Build** a new **REST API** (not HTTP)
- Select **Import from Swagger or Open API 3**
- You can find the Open API specification of our app here: https://caa-lab-templates.s3-eu-west-1.amazonaws.com/openapi.yaml
- Create the API
- Wire the API to the backend
    - Go to API Resources, */products* API, GET method
    - Select HTTP as integration type
    - Paste the load balancer DNS named followed by */api/products*
    - Save. You should be able to test the API from the console (set as headers the following: Authorization: Bearer <your_idToken>)

A New Endpoint

API gateway enables us to add new endpoints that are not necessarily part of the app. We will add a new endpoint for */stocks*. This is intended to be called by our providers after they deliver new products. We will send/buffer all requests to an SQS queue from where a lambda function will read and process them (for now just log them).

First, let's create the SQS queue:

- Go to SQS and press Create queue
- Give it a name and remember it.

Next, let's create the function that will process the messages from the queue:

- Go to Lambda and Create a new function
- Choose Use a blueprint and search for sqs
- You should find sqs-poller. Configure it.
- Give it a name and also provide a name for the role that will be created.
- Choose the SQS queue you created previously and **enable the trigger**
- Glance over the content of the function - it only outputs the received messages.
- Create the function

Before proceeding to extend the API, we have to define an IAM policy and role which will enable API Gateway to send messages to the queue:

- Go to IAM, Policies.
- Create policy
- Choose SQS as service
- Choose SendMessage (under Write) as action
- Under Resources, **add** the **ARN** by providing the region eu-west-1 and your queue name.
- Review, name, and create the policy

- Move to **Roles** and create a new one
- Choose API Gateway as the use case and create the role
- Search for it and attach the policy you created previously

Now, let's create the new API endpoint:

- Go back to your API in API Gateway
- Click on */stocks* and, from Actions, choose Create Method. Select POST as the method from the drop-down.
- Select AWS Service as integration type
- Select eu-west-1 as the region
- Select Simple Queue Service (SQS) as service type
- Select POST as HTTP method
- Change the action type to Use path override and set it to *<aws_account_id>/<queue_name>*
- As the execution role, set the ARN of the role you created previously.
- Save
- Go to **Integration Request**
- Under URL Query String Parameters, add the following:
    - Name: *MessageBody*, Mapped from: *method.request.body.MessageBody*
    - Name: Action, Mapped from: *'SendMessage'* (including the single quotes)
- Go back to the method overview and select Method Response
- Add the HTTP status 200
- Go back to Integration Response and set the HTTP status regex to 200.

Test the New Endpoint

From the method execution page, choose *TEST*. Provide the **request body** as follows:

```
{
   "MessageBody": "well….whatever you want. This will be the content of the message"
}
```

The response should be 200.
Go to the lambda function, under the monitoring tab, and choose View logs in CloudWatch.You should have at least one log stream. Find your message.

The API is not yet reachable by your providers. You have to deploy it. If you reached this point and want to deploy it, you have to add integrations for all methods or to delete them.

Cleanup

As usual, delete everything you created such as:

- The lambda function
- The SQS queue
- The CloudFormation stack
- The API Gateway
- The EC2 instance (if you created one for the AWS CLI)

- The IAM roles and policy
- The Cognito user pool