Cloud Applications Architecture

Lab 1

Intro, Lift & Shift

Connecting via SSH

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.

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.

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

Linux/Mac
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

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>
```

Lift & 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
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. This is nice to have in our case 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 for the lab grade.

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

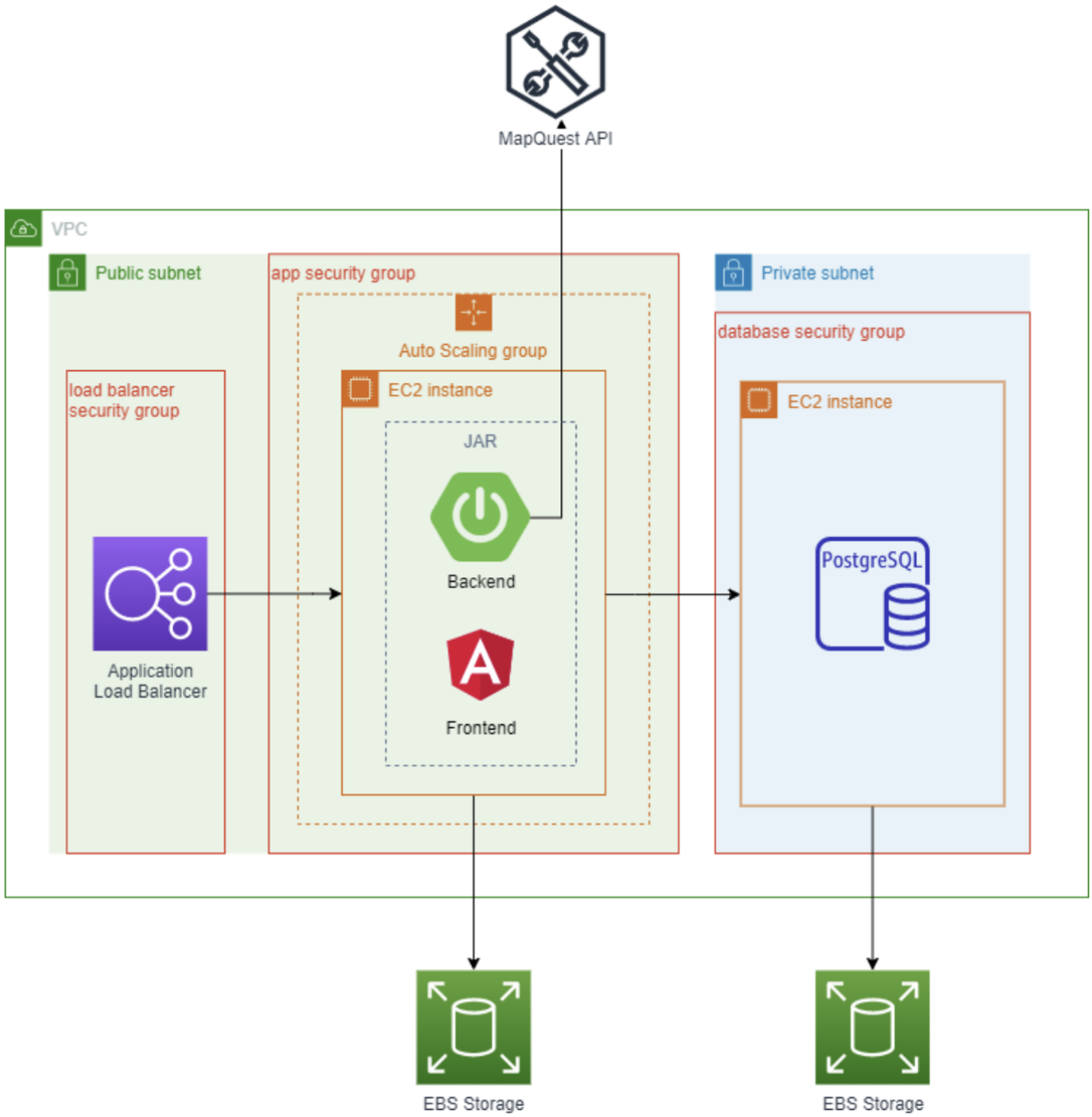
Important: When you stop an instance, the public IP address will be released. When you start the instance again, it will have a new IP address (and DNS name).

Lab 2

High Availability & Scalability

Aim of the Laboratory

- Understand the main **AWS Networking** concepts
- **Separate the deployments** of the app and DB
- Make the application layer **horizontally scalable**



CIDRs, 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 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. Don't worry about IPv6 for now.

You can read more in the AWS VPC Docs.

AWS VPCs

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 a 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.

Basically, VPCs allow you to control the traffic between (sub)networks and use **private IP addresses**.

You can check the AWS VPC Basics docs for more.

Subnets

How IP addresses are constructed makes it relatively simple for Internet routers to find the right network to route data into. However, in a network with many IP addresses, there could be millions of connected devices, and it could take some time for the data to find the right device. Subnetting narrows down the IP address to use 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.

Basically, you will create one or more subnets in a VPC.

Whenever you want to learn more about networking concepts, we recommend the Cloudflare docs. For example, you can learn more about subnets here.

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.

Check out the basics of security groups from the AWS docs.

Load Balancers

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 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 cheaper than an ALB.
- There is a third option, classic, but it's deprecated.
- ***(NEW)* Gateway** - It operates at **layer 3** (network). You can use this if you need to integrate other virtual appliances such as deep packet inspection systems.

Check out the AWS docs.

Auto Scaling Groups

An Auto Scaling Group (ASG) contains a collection of Amazon EC2 instances that are treated as a logical grouping for the purposes of automatic scaling and management. An ASG 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.

Check out the AWS Docs about ASGs.

Lab Walkthrough

Here is what we will achieve:

- Have a dedicated **private subnet** for our database
- Create **strict security groups** for our app and database
- Deploy the database and the application on separate EC2 instances through **CloudFormation**
- Create a **launch template** for the application layer
- Create an **auto scaling group** that will manage the app instances
- Add a **load balancer** in front of the app layer

AWS provides each account with a default VPC out of the box for each region. This VPC has a certain number of subnets, one for each availability zone of the region.

Are the subnets public or private? How can you tell?

The 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.

1. Go to **VPC**
2. From the left navigation pane, go to **Subnets**
3. Press *Create subnet* and configure:
    - **VPC ID** - choose the default VPC
    - **Name** - well, choose a suggestive name. It's a private subnet that we will be using for the database
    - **AZ** - *No preference* should do it
    - **IPv4 CIDR block** - 172.31.64.0/24

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:

1. Go to **EC2**
2. Go to **Security Groups**
3. *Create security group*
    1. **Name**: App-SG
    2. **Description**: Allow application traffic
    3. Add an **inbound rule** to allow traffic on the **port of the app** from all sources
    4. Add the tag **project: caacourse**

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

pin Up the Infrastructure

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):

1. *Template is ready* option should be already selected
2. In the Amazon S3 URL paste the following: https://caa-lab-templates.s3.eu-west-1.amazonaws.com/lab2/infra.yaml
3. Give your stack a name (Lab2 maybe?)
4. Fill in the parameters so CloudFormation configures the instances exactly as we need. Each one of them should have a description.
5. For **DbImageId**, your job is to find it. Hint: It's a private AMI shared with your account.
6. Create the stack (it will take a few minutes)

Until the instances are created and initialized, download the template from above and take a look at it.

In our case, CloudFormation creates the application instance after the database instance is ready.

Why do you think this is necessary?

Creating 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*:

1. Give it a meaningful name
2. Check the Auto Scaling guidance checkbox
3. AMI and instance type should already be set
4. Select the app security group (or make sure it is selected)
5. Remove the network interface (if selected)
6. In the advanced details, set the *Shutdown behavior* and *Stop - Hibernate behavior* to "*Don't include in launch template*".

Creating 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:

1. Go to **Auto Scaling Groups** and press *Create*
2. Give it a meaningful name and select the template you just created
3. On the next step, for the subnet, set at all 3 default public subnets
4. Skip step 3
5. In step 4 set the **desired capacity to 1**, **minimum capacity to 1**, and **maximum capacity to 2**.
6. Skip the notifications, add the usual tag, and create the ASG.
7. 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).

Creating the Load Balancer

The ALB will have a **listener** and a **target group**. The listener is the public part so it should listen on port 80 (HTTP), while the target group should handle the traffic to our application so the port should be set to the app port.

Follow the official guide to create an Application Load Balancer (without HTTPS). Remember to create a **new security group allowing traffic on port 80**.

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

Wrapping Up

Test the app. You should encounter some unexpected behavior. Why does this happen? Can we leverage some settings of the ALB to mitigate the issue?

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 managed by it
- The CloudFormation stack (this should terminate the DB instance)
- EC2 instances if there are any left (shouldn't be)

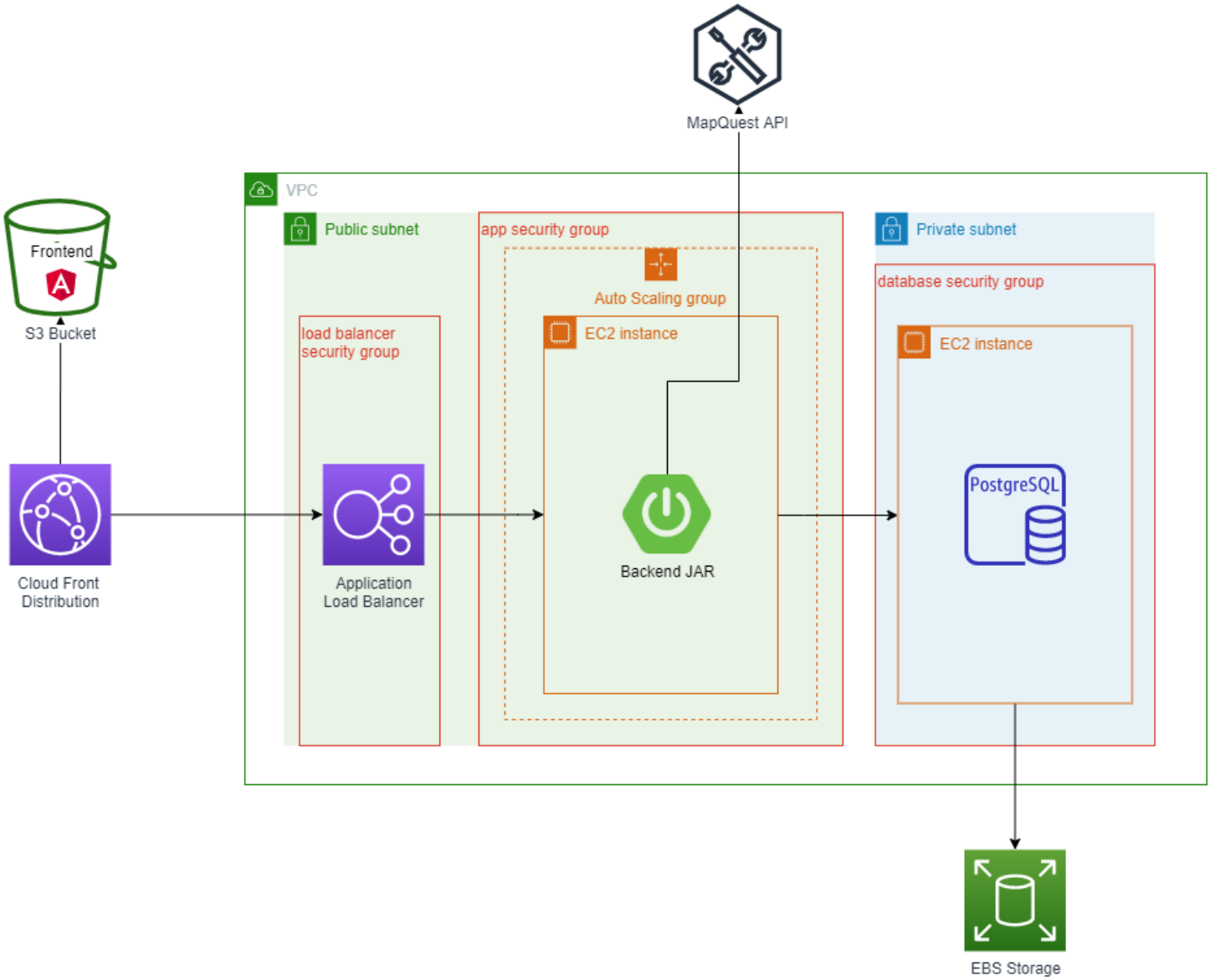
Lab 3

Deploying the Frontend

Aim of the Laboratory

- Familiarize yourself with the frontend of the application

- Connect it to the backend
- Deploy it on AWS
- CDN



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 **frontend**. 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 backend do its job (i.e. process data) and move the presentation layer (frontend) 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 backend (usually through HTTPS requests - AJAX) 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 frontend application separately using suitable AWS services and connect it to our application (which will act just as a backend).

Deploying the Backend

First, we need to deploy the backend, similar to what we did during the previous lab.

You will deploy to CloudFormation stacks - 1 for the DB instance, 1 for the app instance. The templates are at:

1. https://caa-lab-templates.s3.eu-west-1.amazonaws.com/lab2/db.yaml
2. (Make sure the DB instance is ready and shows "2/2 checks passed") https://caa-lab-templates.s3.eu-west-1.amazonaws.com/lab2/app.yaml

This approach is better because, in practice, the database and the application layers have different lifecycles and change at different rates.

If you want to practice setting up the ASG and ALB, give it a try. We should have enough time. Otherwise, you can use the public IP/DNS name of the instance (don't forget the port - 8080)

At this point, you should be able to access the app in the traditional fashion (frontend served by the backend). Credentials: user:storepass

Finding a Suitable AWS Service for the Frontend

AWS offers a huge array of services, some of them being used more than 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 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).

Name 2-3 services based on object storage that most people are using frequently.

Deploying to S3
Your job is to create the S3 bucket, configure it for static hosting, adjust the frontend app so it can reach your backend, and copy the app to the bucket.

Follow the instructions from the AWS docs.

Adjusting the app

- Download the prebuilt angular application from here.
- Search and replace *http://ec2-52-209-147-59.eu-west-1.compute.amazonaws.com:8080* with your backend endpoint address
- Upload all the files to the bucket

Access to S3 buckets can be configured using any of the following methods:

- ACL
- Resource policy
- IAM policy

Identify a use case for each of them.

HTTPS & CDN
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.

Create the CloudFront Distribution
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*

Regarding **Viewer Protocol Policy**, even though we want to redirect all HTTP traffic to HTTPS, doing so will require the backend to also be served over HTTPS, which we don't have for now (we would need a domain). This is a restriction of the browsers (they don't accept plain HTTP if the page was served over 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.

How would you test CloudFront from multiple locations/continents?

Cleanup

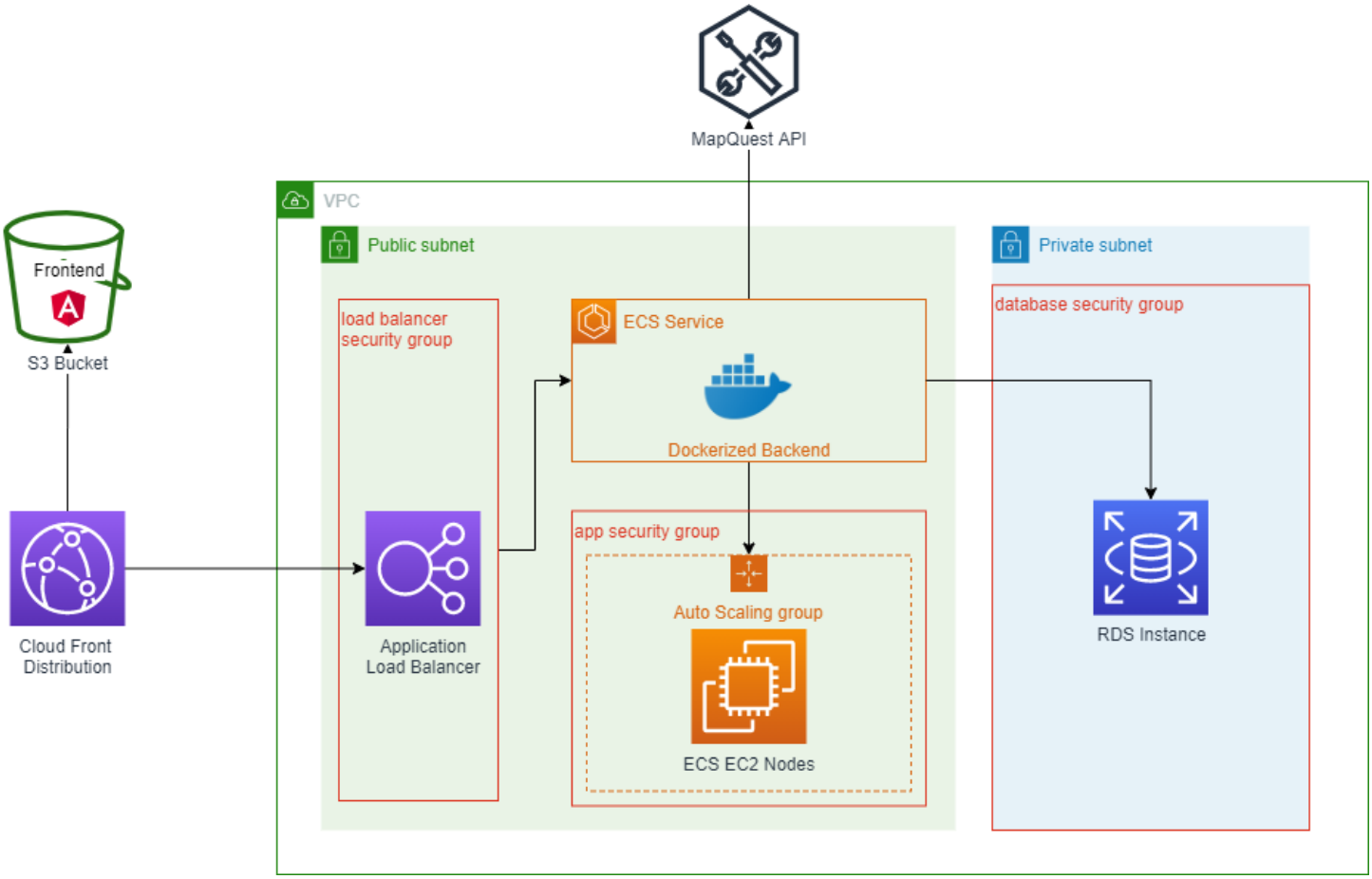
Delete the CloudFormation stacks. Keep the CDN and the S3 bucket.

Lab 4

Managed Services & Containers

Aim of the Laboratory

1. Switching to a managed database
2. Understanding Docker
3. Dockerizing our app
4. Finding and deploying to a suitable AWS service



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 instance's health 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 and applying updates and backups.

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

Creating an RDS Instance

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 which is more than enough for our needs.
3. Give it a name and set the password to **postgres**.
4. For DB instance size, db.t3.micro should be set automatically.
5. For storage, 20GB of general-purpose SSD should suffice and no autoscaling needed.
6. For the **connectivity** section, make sure that you configure it the same as the DB instance created in lab 2. (use the database security group you previously created).
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 and enhanced monitoring.
8. It will take a few minutes until the RDS instance is ready.

Deploying the App Tier

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

Updating the Frontend

Follow the steps from the previous lab to deploy the frontend application. Update the load balancer endpoint with the new one.

Docker Intro

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).

You can always find out more from the Docker documentation.

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 out 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 containers)

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 --name first my_first_image*

Cleanup

Remove the hello-world container and the images

- *docker rm first*
- *docker rmi my_first_image*

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 ENTRYPOINT, you can use the following line (update the RDS endpoint):

ENTRYPOINT ["java", "-Dspring.profiles.active=with-form", "-DPOSTGRES_HOST=rds-lab4.ckcr0x2dmlbu.eu-west-1.rds.amazonaws.com", "-DPOSTGRES_PORT=5432", "-DPOSTGRES_USER=postgres", "-DPOSTGRES_PASSWORD=postgres", "-jar", "/app.jar"]

Publishing the Image

Now, you will push the image to a container registry - AWS ECR in this case:

1. Create a registry from the console
2. From the terminal, authenticate on ECR with:
    1. *export AWS_DEFAULT_REGION=eu-west-1*

2. *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 the official guide to actually push the image

Deploy the Container

Create an ECS cluster

An ECS cluster uses EC2 instances as the 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)

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 that 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.

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
- 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

Deregister the Previous App Instance

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. Deregister the one from the beginning.

Cleanup

Delete the following:

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

Lab 5

IAM & Recap

IAM Concepts

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. Ideally, each user has a certain set of permissions adhering to the least privilege principle and segregation/separation of duties.

Groups

Groups are what you probably expect to be - groups of users. The main benefit is that groups allow us to maintain the permissions for multiple users (the members of a group). This is an important best practice.

In AWS, there are no predefined groups (just like there are no predefined users).

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 and service control policies (SCPs) that are applied at the organization level.

Of course, you can always check the AWS documentation here. Besides some intimidating blocks of text 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.

Roles can also be assumed by resources such as EC2 instances. This is the preferred way to provide permissions to your applications. If we didn'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) which is a bad practice.

When would a resource-based policy be better suited than an identity-based policy?

Hands-On with Users, Policies, and Roles

Your user has full admin rights. However, in real-life scenarios, this will not be the case. Let's create a more suitable setup.

Create an IAM admin role

1. Go to IAM (not IAM Identity Center)
2. Go to roles
3. Create a new role that users from **your AWS account** can assume
4. Attach the *IAMFullAccess* policy
5. Name it **IAMAdminRole**
6. Click Create

Managing IAM

In practice, access management is a highly-privilege action. Ideally, when admins must make any changes in IAM (create users, change permissions, etc), they should elevate their permissions for a short time span. Also, these actions must be logged for audit purposes. This is an excellent use case for AWS IAM roles.

Let's say that you are the administrator of your organization. A new developer, John, joins your team, so you must create an AWS user for him. He needs full access to EC2 and read access to RDS

1. Assume the role you just created (copy the account ID from the top-right menu, you will need it)
2. Go to IAM groups
3. Create a new group called *Developers* with the right policies attached (*AmazonEC2FullAccess* and *AmazonRDSReadOnlyAccess*)
4. Create a new user, John, and assign him to the new group.
5. If you want, you can log in as John in a separate browser or incognito window

IAM vs IAM Identity Center

Confusingly enough, AWS provides two services for IAM:

1. IAM - the one we practiced today
2. IAM Identity Center - the one through which you access your AWS account. Previously it was called AWS SSO.

Ideally, you would always use IAM Identity Center. It's better at managing access to multiple AWS accounts (it's a common practice to have applications spanning multiple AWS accounts). In practice, the world still heavily relies on the good old IAM.

Cleanup

Delete the IAM resources you just created (user, group, role).

EXERCITII

The AWS service for load balancing supports the following types of load balancers:

- Application load balancer which operates at ✓OSI Layer 7 (HTTP/WebSockets)
- Network load balancer which operates at ✓ OSI Layer 4 (TCP/UDP)

An AWS Region is*: (answer not sure)*

between: ✓ An organizational unit designed to serve clients from a specific area
An independent and isolated cloud deployment in a specific geographical region

Simple Storage Service (S3) helps us: ✓ Store and serve an infinite amount of objects such as images, documents, and videos

RDS is a: ✓ Managed relational database service

Documents are stored in S3 in a: ✓ Bucket

An EC2 instance can be configured as follows:
✓ Based on an instance type which offers certain resources (vCPU, RAM, networking, etc.)

Software can be automatically installed during the provisioning of the EC2 instance through ? something (but not *User Data*) which is composed of ✓ A CloudFormation script

Simple Storage Service works through: ✓ HTTP

An EC2 instance is deployed in a specific AZ: ✓ True

Docker is a: ✓ Container engine

Content stored on S3 can be restricted such that it's accesible only through CloudFront by using: ✓ Origin Access Identity

Your application tier is running in an Auto Scaling Group and you need to change the instance type. In which of the following area can this be achieved? ✓ Auto Scaling launch template/configuration

Objects storedd in S3 can be made public by configuring the ? something (but not *S3 Settings*)
? something (but not *after the objects are uploaded*)

Your organization is running a business-critical learning management system (LMS) on its own infrastructure on-premises. Since the platform has been growing exponentially, the stakeholders decided to migrate it to AWS. You are part of the development team and you are responsible for choosing the right services and designing the solution. The system enables teachers and experts to market, sell, and present their courses to people willing to learn from all around the world. Each course might contain written documents (mostly PDFs and PowerPoint presentations), videos, and images/diagrams. Moreover, students can be notified via email, each course goes through a review process before being published, and teachers have access to the API of the system enabling them to publish the courses on their own websites.

The LMS is composed out of several backend modules written in Java and Node.js which store data in MySQL databases, while the frontend is built with React. Your team has been struggling lately to run the system locally due to its distributed nature and increasing number of dependencies. Another important consideration is that the system transcodes the videos before the course is published in order to optimize bandwidth and playback. Since this is the core product of your organization, the main focus of the design must be the availability of the system and reducing operational/maintenance overhead. Of course, cost must be optimized whenever possible. Describe your solution below. Mention which AWS services you would choose, how they communicate to each other, and, equally important, why you have chosen a particular service. Be as explicit as possible. Optionally, you can upload images to illustrate your design (feel free to use any drawing or diagram tool you wish; images don't have to be pretty).