Cloud Applications Architecture Labs

Lab 1 - Intro to AWS

AWS Regions, AZs

Amazon cloud computing resources are hosted in multiple locations worldwide. 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.

The AWS Console

The AWS Console is a management interface that acts as a central entry point for 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.

Lab 1 - Questions

- DynamoDB - NO SQL Amazon database
- S3 - a massively scalable storage service based on object storage technologies
- FaaS (function as a service) - AWS Lambda → allows you to run serverless functions in response to events like HTTP requests

Deploying a VM

Regions

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).

Make sure you are in the Ireland (eu-west-1) region.

Create an EC2 Instance

We will now create our first AWS virtual machine. Navigate to the EC2 service and, in the instances section, select **Launch Instance**.

Tags

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

For EC2 (and not only), AWS uses tags to store the name of the resource. So, besides the name (suggestion: *caa-lab1*), add the additional 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.**

AMIs

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.

For the AMI (Amazon Machine Image), use Amazon Linux 2023 (should be the default). This AWS-optimized Linux flavor can be used as part of the AWS free tier. It is configured by AWS and comes with several packages pre-installed such as the AWS CLI.

Instance Type

Amazon EC2 provides a wide selection of instance types optimized for 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 **t2.micro** (or t3.micro if not available). This provides a machine with 1 vCPU and 1 GB of memory. It is more than enough for our needs. Also, it is a burstable machine type.

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.

Network settings

We will connect to this instance via SSH, so it's 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 plain HTTP, allow incoming connections on that. (you might have to press **Edit**)

Storage

The default configuration should include an 8GB EBS volume (network storage) — no need to change anything here.

Advanced details

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.). You can find the script on the next page.

Connecting via SSH

The 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(WSL)/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:

curl -L -O https://github.com/CAA-Course/app/releases/download/1.0/shop-1.0.jar
sudo java -Dspring.profiles.active=with-form,local -Dserver.port=80 -jar shop-1.0.jar

Note: Sudo is required for running applications on ports below 1024 (these are called privileged ports). The optimal approach would be to install a web server/reverse proxy such as Apache httpd or Nginx that will serve our application running on other ports such as 3000 or 8080.

Your application should now be accessible at the above-mentioned public IP address or DNS name. 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).

Lab 2 - High Availability & Scalability

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

Due to how IP addresses are constructed, it is relatively simple for 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 (packet) 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 indicate which subnet an IP packet should go to. Routers within a network use a **subnet mask** to sort data into subnetworks.

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

We recommend the Cloudflare docs whenever you want to learn more about networking concepts. 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.

Tip: if the connection to/from your instance times out, the most likely cause is the security group.

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 (ELB), 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 2 - Questions

- 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. Inspect the default VPC from your account and answer the following questions:
  1. What is the IPv4 CIDR range of the VPC? ([first IP] - [last IP])
  2. What subnets are available by default? Are they public or private?
- Why do you think we deploy the DB and the app in separate CloudFormation stacks?

Walkthrough

The Private Subnet

A subnet is a subrange 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 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. If this doesn't work, there is already a private subnet. You can reuse that.

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*
    a. **Name**: App-SG
    b. **Description**: Allow application traffic
    c. Add an **inbound rule** to allow traffic on the **port of the app** from all sources
    d. 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**.

Spin 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):

You will create two stacks: one for the DB, one for the app

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/db.yaml
3. Give your stack a name (suggestion: "Lab2-DB")
4. Fill in the parameters so CloudFormation configures the instances precisely as we need them.
5. For **DbImageId**, your job is to find it. Hint: It's a private AMI shared with your account. (make sure you are in Ireland)
6. Create the stack (it will take a few minutes)
7. When the DB EC2 is ready, repeat the steps to deploy the app instance using the S3 URL (**different template**) https://caa-lab-templates.s3.eu-west-1.amazonaws.com/lab2/app.yaml. It should ask for the database's private IP address. You can get it from EC2 or the outputs of the first CloudFormation stack.

When everything is ready, you should be able to access the app using the public IP/DNS name of the EC2 instance (plain HTTP). Until the instances are created and initialized, download the templates from above and look at them.

Creating the Launch Template

Launch templates 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. For the subnet, select "*Don't include in launch template*" and remove the network interface.
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). The ASG should manage two instances now (which is why we chose a maximum capacity of 2).

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 forwards the traffic to our application so the port should be set to the app port (again, 80).

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.

Lab 3 - Deploying the Frontend

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 - one for the DB instance and one for the app instance. The templates are at:

1. Practice recreating the **security groups** (remember, the database is Postgres, and the app listens on port 80)
2. https://caa-lab-templates.s3.eu-west-1.amazonaws.com/lab2/db.yaml
3. Make sure the DB instance is **ready**
4. https://caa-lab-templates.s3.eu-west-1.amazonaws.com/lab2/app.yaml

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.

At this point, you should be able to access the app just like before (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).

Lab 3 - Questions

- Name 2-3 services based on object storage that most people are using frequently.
- Access to S3 buckets can be configured using any of the following methods:

  Identify a use case for each of them.

  - ACL
  - Resource policy
  - IAM policy
- How would you test CloudFront from multiple locations/continents?

Deploying to S3

Your job is to:

1. Create the S3 bucket
2. Configure it for static hosting
3. Adjust the frontend app to reach your backend
4. Copy the app to the bucket.

Follow the instructions in the AWS docs. You should also follow the steps to adjust the permissions of your bucket.

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. Ensure you add "http://".
- Upload all the files to the root of the bucket (ensure to upload just the files, not the folder)

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***
- Origin access: **Origin access controls settings**
- **Create new OAC**. You will have to update the bucket's policy (make sure you open the bucket in Ireland)
- WAF: **No**
- 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.

Lab 4 - Managed Services & Containers

Intro

Moving back to our backend application, in this lab, we will focus on reducing the stuff that we manage. Until now, we have had to manage the ec2 instance where our application runs 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 move our database to a managed service – RDS (Relational Database Service).

In the second part, we will focus on making our application simpler to deploy and run using Docker and migrating it to a suitable AWS service.

Network Setup

Use **CloudFormation** to create a new VPC with private/public subnets and the required security groups.

The template is: https://caa-lab-templates.s3.eu-west-1.amazonaws.com/lab4/vpc.json

RDS

RDS is a managed database service. We tell AWS the hardware specs and database engine we need (and some other info – mainly security and availability), and it will run and manage the database for us. This means we no longer 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 (Postgres) as we used in previous labs.

Bonus tip: In a real-world scenario, we would migrate the data from our self-managed database to the RDS instance. We can achieve this using utilities such as pg_dump (guide from AWS) or managed services such as the AWS Database Migration Service.

Creating an RDS Instance

1. Go with the “standard create” mode instead of “easy create”. This will give you a better glimpse into the capabilities of RDS. Choose **PostgreSQL** (not Aurora).
2. **Template**: Free tier. This is more than enough for our needs.
3. Give it a name and set the password to "postgres**"** (the app expects it like this).
4. For DB instance size, db.t3.micro should be set automatically.
5. For storage, 20GB of general-purpose SSD should suffice (switch to gp3). No autoscaling is needed.
6. In the **connectivity** section, "Don't connect to an EC2 instance", choose the newly created VPC, no public access, and choose the "db-sg" security group.
7. Under **additional configuration**, set the **initial database name** to "postgres". This is important because otherwise, RDS wouldn’t create our database. Also, disable automatic backups and enhanced monitoring.
8. It will take a few minutes until the RDS instance is ready.

Deploying the App Tier

Before you do this, ensure the RDS instance is **Available** (it takes a few minutes).

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

For the subnets parameter, select only the public subnets (there should be two).

Docker Intro

Docker is a container engine. A container is an isolated environment where programs can be run. 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 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 about it in the Docker documentation.

We will use our app's EC2 instance to better understand Docker and "dockerize" our application. Another option would be to install the Docker engine locally.

Install Docker

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

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 the Containers on Your Machine

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

Build Your Own Container Image

Creating a new Docker image means telling Docker the following:

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

This is all achieved using the **Dockerfile**.

Create your 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 create
d
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*

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.

Create a new Dockerfile in the same directory as the jar. Use this Dockerfile, replace the POSTGRES_HOST parameter with your RDS endpoint, and build the image with *docker build -t shop:latest .*

Publishing the Image

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

1. Create a private repository from the console
2. Follow the steps from "View push commands"

Deploy the Container

Create an ECS cluster

While creating a cluster based on EC2 instances would be more insightful, for simplicity we will go with **Fargate** (AWS will manage the compute layer on our behalf).

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 to 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, ensuring that a predefined number of container instances is always available. The service also exposes your container through the load balancer.

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

- Launch Type: Fargate
- 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

Delete the initial target group from the load balancer (the one pointing to the EC2 instance)

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.

Lab 5 - Questions

- 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.