Bootcamp
Data Engineering

Module02
Cloud API

Module02 - Cloud Storage API

In the module, you will learn how to use a Cloud Provider. For all the exercises, I took Amazon Web Services (AWS) as an example but **you are totally free to use the cloud provider you want which is compatible with Terraform** (we advise you to use AWS if you don't have one). AWS has become the most popular cloud service provider in the world followed by Google Cloud Platform and Microsoft Azure.

Amazon Web Services started in 2005 and it now delivers nearly 2000 services. Due to the large number of services and the maturity of AWS, it is a better option to start learning cloud computing.

If you never heard about the Cloud before, do not worry! You will learn step by step what the Cloud is and how to use it.

Notions of the module

The module will be divided into two parts. In the first one, you will learn to use a tool called Terraform which will allow you to deploy/destruct cloud infrastructures. In the second part of the module, you will learn to use a software development kit (SDK) which will allow you to use Python in order to interact with your cloud.

General rules

- The exercises are ordered from the easiest to the hardest.
- Your exercises are going to be evaluated by someone else, so make sure that your variable names and function names are appropriate and civil.
- Your manual is the internet.
- You can also ask any question in the dedicated channel in Slack: **42ai slack**.
- If you find any issue or mistakes in the subject please create an issue on our dedicated repository on Github: **Github issues**.

Foreword

Cloud computing is the on-demand delivery of IT resources and applications via the Internet with pay-as-you-go pricing. In fact, a cloud server is located in a data center that could be anywhere in the world.

Whether you run applications that share photos to millions of mobile users or deliver services that support the critical operations of your business, the cloud provides rapid access to flexible and low-cost IT resources. With cloud computing, you don't need to make large up-front investments in hardware and spend a lot of time managing that hardware. Instead, you can provision exactly the right type and size of computing resources you need to power your newest bright idea or operate your IT department. With cloud computing, you can access as many resources as you need, almost instantly, and only pay for what you use.

In its simplest form, cloud computing provides an easy way to access servers, storage, databases, and a broad set of application services over the Internet. Cloud computing providers such as AWS own and maintain the network-connected hardware required for these application services, while you provision and use what you need for your workloads.

As seen previously, Cloud computing provides some real benefits :

- **Variable expense**: You don't need to invest in huge data centers you may not use at full capacity. You pay for how much you consume!
- **Available in minutes**: New IT resources can be accessed within minutes.
- **Economies of scale**: A large number of users enables Cloud providers to achieve higher economies of scale translating at lower prices.
- **Global in minutes**: Cloud architectures can be deployed really easily all around the world.

Deployments using the cloud can be `all-in-cloud-based` (the entire infrastructure is in the cloud) or `hybrid` (using on-premise and cloud).

AWS global infrastructure

Amazon Web Services (AWS) is a cloud service provider, also known as infrastructure-as-a-service (`IaaS`). AWS is the clear market leader in this domain and offers much more services compared to its competitors.

AWS has some interesting properties such as:

- **High availability** : Any file can be accessed from anywhere
- **Fault tolerance**: In case an AWS server fails, you can still retrieve the files (the fault tolerance is due to redundancy).
- **Scalability**: Possibility to add more servers when needed.
- **Elasticity**: Possibility to grow or shrink infrastructure.

AWS provides a highly available technology infrastructure platform with multiple locations worldwide. These locations are composed of `regions` and `availability zones`.

Each region represents a unique geographic area. Each region contains multiple, isolated locations known as availability zones. An availability zone is a physical data center geographically separated from other availability zones (redundant power, networking, and connectivity).

You can achieve high availability by deploying your application across multiple availability zones.

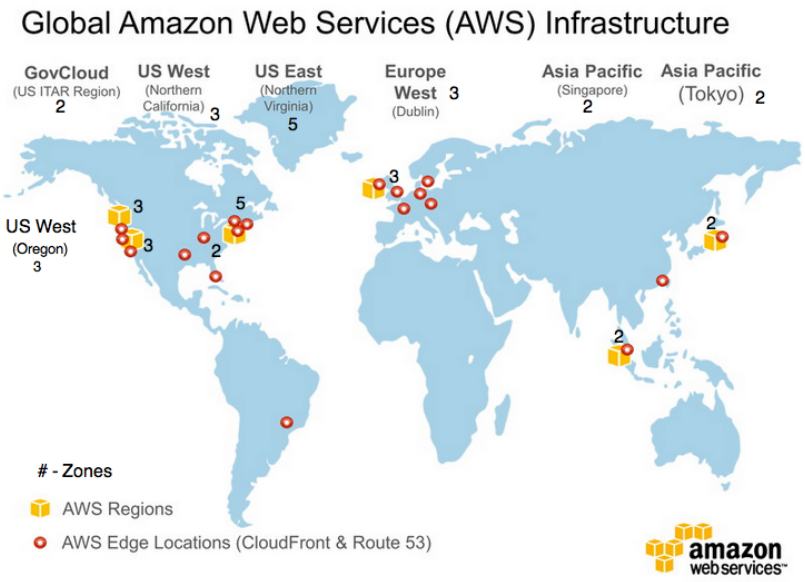

Figure 1: AWS regions

The `edge locations` you see on the picture are endpoints for AWS which are used for caching content (performance optimization mechanism in which data is delivered from the closest servers for optimal application performance). Typically consists of CloudFront (Amazon's content delivery network (CDN)).

Helper

- Your best friends for the module: **AWS documentation** and **Terraform documentation**.

Exercise 00 - Setup

Turn-in directory:	ex00
Files to turn in:	
Forbidden function:	None
Remarks:	n/a

In this exercise we are going to setup our account to start working with a cloud provider.

Don't worry! Even if you enter your card number, this module should not cost you anything. First, indeed, AWS as a free tier usage (check if it the case also for your cloud provider) that allows you to use a small amount of AWS resources for free. This will be sufficient enough for what you are going to do today. By the end of the day, you will have to entirely destroy your infrastructure (don't keep things running) !!!

Exercise

- Create an account on your cloud provider (all the exercise were made using AWS but you can choose another cloud provider).
- Set up a billing alarm linked to your email that will alert you if the cost of your infrastructure exceeds 1$.
- Create a new administrator user separated from your root account (you will need to use this user for all the exercises). Save the credentials linked to the administrator user into a file called `credentials.csv`.

All the mechanisms we are creating now will ensure your access is secured and will allow you to quickly be alerted if you forgot to destroy your infrastructure.

Exercise 01 - Storage

Turn-in directory:	ex01
Files to turn in:	presigned_url.sh
Forbidden function:	None
Remarks:	n/a

AWS CLI

We are going to use the AWS command-line interface. The first thing we need to do is install it.

You should be able to run `aws --version` now.

We can setup our AWS account for the CLI with the command `aws configure`. You will need to enter:

- access key : in your `credentials.csv` file
- secret access key : in your `credentials.csv` file
- region : `eu-west-1` (Ireland)
- default output format : `None`

The AWS CLI is now ready!

S3 bucket creation

Amazon S3 provides developers and IT teams with secure, durable, and highly-scalable cloud storage. Amazon S3 is easy-to-use object storage with a simple web service interface that you can use to store and retrieve any amount of data from anywhere on the web.

A bucket is a container (web folder) for objects (files) stored in Amazon S3. Every Amazon S3 object is contained in a bucket. Buckets form the top-level namespace for Amazon S3, and bucket names are global. This means that your bucket names must be unique globally (across all AWS accounts). The reason for that is when we create a bucket, it is going to have a web address (ex : `https://s3-eu-west-1.amazonaws.com/example`).

Even though the namespace for Amazon S3 buckets is global, each Amazon S3 bucket is created in a specific region that you choose. This lets you control where your data is stored.

With your free usage you can store up to 5 Gb of data!

Exercise

In this exercise, you will learn to create an S3 bucket and use aws-cli.

- Connect to the console of your administrator user
- Create an S3 bucket starting with the prefix `module02-` and finished with whatever numbers you want.
- Using aws-cli, copy `appstore_games.csv` file to the bucket. You can check the file was correctly copied using the AWS console.
- Using aws-cli, create a presigned URL allowing you to download the file. Your presigned url must have an expiring time of 10 minutes. Your AWS CLI command must be stored in the `presigned_url.sh` script.

Exercise 02 - Compute

Turn-in directory:	ex02
Files to turn in:	os_name.txt
Forbidden function:	None
Remarks:	n/a

Amazon Elastic Compute Cloud (Amazon EC2) is a web service that provides resizable compute capacity in the cloud. Amazon EC2 reduces the time required to obtain and boot new server instances to minutes, allowing us to quickly scale capacity (up or down) depending on our needs.

Amazon EC2 allows you to acquire compute through the launching of virtual servers called instances. When you launch an instance, you can make use of the compute as you wish, just as you would with an on-premises server (local servers). Because you are paying for the computing power of the instance, you are charged per hour while the instance is running. When you stop the instance, you are no longer charged.

Two concepts are key to launching instances on AWS:

- **instance type** : the amount of virtual hardware dedicated to the instance.
- **AMI (Amazon Machine Image)** : the software loaded on the instance (Linux, MacOS, Debian, ...).

The instance type defines the virtual hardware supporting an Amazon EC2 instance. There are dozens of instance types available, varying in the following dimensions:

- Virtual CPUs (vCPUs)
- Memory
- Storage (size and type)
- Network performance

Instance types are grouped into families based on the ratio of these values to each other. Today we are going to use t2.micro instances (they are included in the free usage)!

One of the impressive features of EC2 is autoscaling. If you have a website, with 100 users you can have your website running on a little instance. If the next day, you have 10000 users then your server can scale up by recruiting new ec2 instances to handle this new load!

Exercise

In this exercise, you will learn how to create and connect to an ec2-instance. If you are on another cloud provider aim for linux based instances with a very little size (if it can be free it is better).

Follow these steps for the exercise:

- launch an ec2 instance with the AMI : `Amazon Linux 2 AMI`.
- choose `t2.micro` as instance type.
- create a key pair.
- connect in ssh to your instance using your key pair.
- get and save the os name of your instance in the `os_name.txt` file.
- terminate your instance.

Within minutes we have created a server and we can work on it!

Exercise 03 - Flask API - List & Delete

Turn-in directory:	ex03
Files to turn in:	app.py, *.py
Forbidden function:	None
Remarks:	n/a

Before getting into AWS infrastructure, we are going to discover how to interact with AWS resources using a Python SDK (Software Development Kit) called boto3. We are going to work with a python micro framework called Flask to create an API (a programmatic interface) to interact with your s3 bucket. For now, the API will be built locally to ease the development.

nb: for a simplification of the following exercises we are going to use Flask directly like a development environment. If we wanted a more production-ready application we would add a webserver like Nginx/Apache linked with Gunicorn/Wsgi.

Exercise

Create a Flask application `app.py` with three routes:

- `/`
  - **status** : `200`
  - **message** : `Successfully connected to module02 upload/download API`
- `/list_files` :
  - **status** : `200`
  - **message** : `Successfully listed files on s3 bucket '<bucket_name>'`
  - **content** : list of files within the s3 bucket
- `/delete/<filename>` :
  - **status** : `200`
  - **message** : `Successfully deleted file '<filename>' on s3 bucket '<bucket_name>'`

The content you return with your Flask API must be json formatted. You should use boto3 to interact with the s3 bucket you previously created (`module02-...`).

Exercise 04 - Flask API - Download & Upload

Turn-in directory:	ex04
Files to turn in:	app.py, *.py
Forbidden function:	None
Remarks:	n/a

We will continue to work on our Flask API to add new functionalities. This time we will work around file download and upload. In order to upload and download files we are going to use something we already used, presigned urls!

Exercise

Create a Flask application `app.py` with two more routes:

- `/download/<filename>` :
  - **status** : `200`
  - **message** : `Successfully downloaded file '<filename>' on s3 bucket '<bucket_name>'`
  - **content** : presigned url to download file
- `/upload/<filename>` :
  - **status** : `200`
  - **message** : `Successfully uploaded file '<filename>' on s3 bucket '<bucket_name>'`
  - **content** : presigned url to upload file

The content you return with your Flask API has to be json formatted. You should use boto3 to interact with the s3 bucket you previously created (`module02-...`).

Exercise 05 - Flask API - Download

Turn-in directory:	ex05
Files to turn in:	client.py app.py, *.py
Forbidden function:	None
Remarks:	n/a

Our API is finished but it is not quite convenient to request! To ease the use of this API, we are going to create a client that will allow us to interact more easily with the API.

Exercise

Create a client `client.py` that will call the API you are creating and show results in a more human readable way. The client will have two parameters:

- `--ip`: IP address of the API (the default IP must be defined as `0.0.0.0`)
- `--filename`: file name to delete, download or upload.

... and the following options:

- `ping`: call the route `/` of the API and print the message.
- `list`: call the route `/list_files` of the API and show the files on the bucket.
- `delete`: call the route `/delete/<filename>` of the API and delete a file on the bucket.
- `download`: call the route `/download/<filename>` of the API and download a file from the bucket.
- `upload`: call the route `/delete/<filename>` of the API and upload a file on the bucket.

Exercise 06 - IAM role

Turn-in directory:	ex06
Files to turn in:	00_variables.tf, 01_networking.tf, 07_iam.tf, 10_terraform.auto.tfvars
Forbidden function:	None
Remarks:	n/a

Terraform is a tool to deploy infrastructure as code. It can be used for multiple cloud providers (AWS, Azure, GCP, ...). We are going to use it to deploy our new API!

As you already know, we are using our AWS free tier. However, if you let your server run for weeks you will have to pay. We want to avoid this possibility. That's why we are going to use a tool to automatically deploy and destroy our infrastructure, Terraform.

All potentially critical data **MUST NOT** be deployed using infrastructure as code like terraform. If they are, they may be destroyed accidentally and you never want that to happen!

Terraform install

First, download the terraform software for macOS.

```
brew install terraform
```

You can now run the `terraform --version`. Terraform is ready!

Terraform is composed of three kinds of files:

- `.tfvars` : terraform variables.
- `.tf` : terraform infrastructure description.
- `.tfstate` : describe all the parameters of the stack you applied (is updated after an apply)

You can run `terraform destroy` to delete your stack.

No further talking, let's deep dive into Terraform!

For all the following exercises, all the resources that can be tagged must use the `project_name` variable with the following tags structure:

- `Name: <project_name>-<resource_name>`
- `Project_name: <project_name>`

Variables must be specified in variable files!

Exercise

For this first exercise, you will have to use the default VPC (Virtual Private Cloud). A VPC emulates a network within AWS infrastructure. This default VPC ease the use of AWS services like EC2 (you do not need to know anything in network setup). You will have to work in the Ireland region (this region can be changed depending on the cloud provider and your location).

The main objective is to create an IAM role for an EC2 instance allowing it to use all actions on s3 buckets (list, copy, ...). In order to create a role in terraform you will have to create:

- a role called `module02_s3FullAccessRole`
- a profile called `module02_s3FullAccessProfile`
- a policy called `module02_s3FullAccessPolicy`

To test your role you can create an EC2 instance and link your newly created role to it, if the AWS cli works then the exercise is done. You must be able to destroy your stack entirely.

Exercise 07 - Security groups

Turn-in directory:	ex07
Files to turn in:	08_security_groups.tf, *.tf, *.auto.tfvars
Forbidden function:	None
Remarks:	n/a

As you already noticed, Flask uses the port 5000. In EC2 all the incoming and outgoing traffic is blocked by default (for security reasons). If we want to interact with our API we will have allow the traffic. In AWS, we can define traffic rules using security groups. The security group will then be associated with an EC2 instance.

Exercise

Create a security that will allow:

- `ssh` incoming traffic (we will use it in the next exercise)
- `tcp` incoming traffic on port `5000` (to interact with our Flask API)
- outgoing traffic to the whole internet

To test the security group you can associate it to a newly created EC2 instance (you will need to use an existing key pair).

Exercise 08 - Cloud API

Turn-in directory:	ex08
Files to turn in:	02_ec2.tf, *.tf, *.auto.tfvars
Forbidden function:	None
Remarks:	n/a

As you may have noticed, building a whole infrastructure needs a lot of steps (which includes). To ease the deployment, it was split into two parts: an intermediate deployment and the final implementation. The first part consists in deploying one EC2 instance with your working API. Thus we will ensure everything we did on terraform is working well. I have a good news, by the end of this exercise you will have done the first intermediate solution!

Exercise

To finalize our intermediate infrastructure we are going to add two components.

First, you will need a key-pair file we will call `module02.pem` and provisioned through Terraform. The key pair must use the RSA algorithm and have the appropriate permissions to be functional.

You must provision an EC2 resource which will use:

- the default vpc
- the role you created
- the security group you created
- the key pair you just created
- a public ip address
- an instance type `t2.micro` with a Linux AMI

You must create an output that will show the public ip of the instance you created.

At this point you should be able to ssh into the EC2 and use aws cli on s3 buckets. However, our API is still not working!

First, upload the files of your API onto your s3 bucket (you don't need to upload the client). Those files should never be deleted to provision your API. This solution is not the cleanest solution but it will be sufficient for the purpose of this module.

Create a bootstrap script that will:

- install the necessary libraries
- download the files of the API from the s3 bucket
- start the API in background

The exercise will be considered valid only if the API is working after a `terraform apply`. You should be able to use your client on the output ip.

Exercise 09 - Network

Turn-in directory:	ex09
Files to turn in:	00_variables.tf, 01_networking.tf, 10_terraform.auto.tfvars
Forbidden function:	None
Remarks:	n/a

AWS and I lied to you! You thought deploying a server was that simple? A huge part of the required stack for the deployment is hidden! This hidden layer uses a wizard configuration (default configuration suitable for most users). The default configuration includes:

- network (VPC, subnets, CIDR blocks)
- network components (routing table, Internet gateway, NAT gateway)
- security (NACLS, security groups)

Exercise

For this new implementation we are going to recode more parts of our architecture like the network. We are going to use our own VPC to not rely on the default one.

Create a VPC using terraform. You have to respect the following constraints:

- your vpc is deployed in Ireland (specified as the variable `region`).
- your vpc uses a `10.0.0.0/16` CIDR block.
- your vpc must enable DNS hostname (this will be useful for the next exercises)

On your AWS console, you can go in the VPC section to check if your VPC was correctly created.

Within our newly created VPC we want to divide the network's IPs into subnets. This can be useful for many different purposes and helps isolate groups of hosts together and deal with them easily. In AWS, subnets are often associated with different availability zones which guarantees the high availability in case an AWS data center is destroyed.

Within our previously created VPC, add 2 subnets with the following characteristics:

- their depends on the creation of the VPC (it has to be specified on terraform)
- your subnets will use `10.0.1.0/24` and `10.0.2.0/24` CIDR blocks.
- your subnets will use `eu-west-1a` and `eu-west-1b` availability zones.
- they must map public ip on launch.

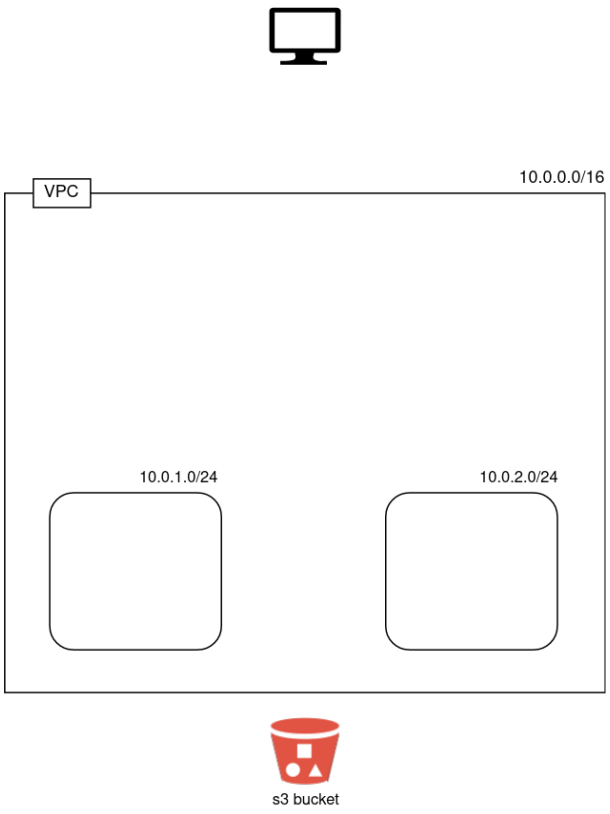

Figure 2: Flask API AWS infrastructure

Exercise 10 - IGW - Route table

Turn-in directory:	ex10
Files to turn in:	00_variables.tf, 01_networking.tf, 10_terraform.auto.tfvars
Forbidden function:	None
Remarks:	n/a

Let's continue our infrastructure! We created a Network with VPC and divided our network into subnets into two different availability zones. However, our network is still not accessible from the internet (or your IP). First, we need to create an internet gateway and link it with our VPC. This first step will allow us to interact with the internet.

The subnets we created will be used to host our EC2 instances but our subnets are now disconnected from the internet and other IPs within the VPC. To fix this problem, we will create a route table (it acts like a combination of a switch (when you interact with IPs inside your VPC) and a router (when you want to interact with external IPs)).

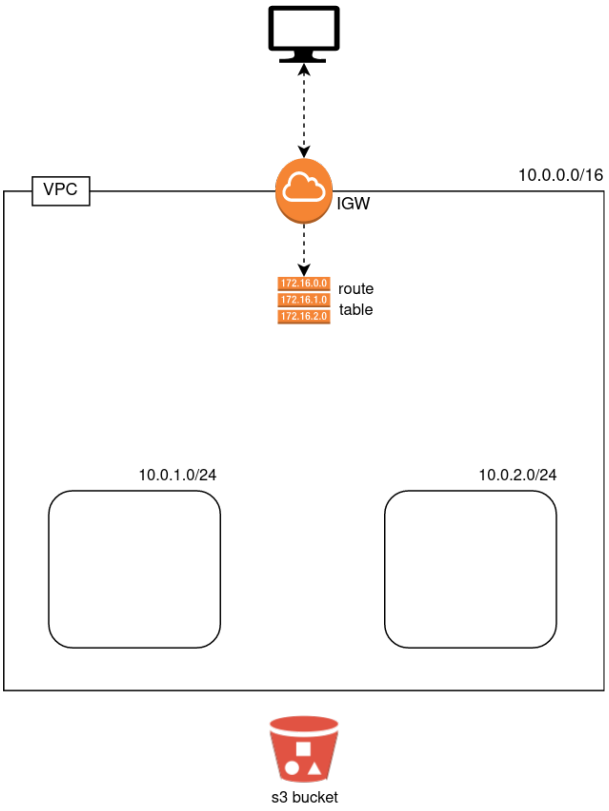

Figure 3: Flask API AWS infrastructure

Exercise

Create an Internet gateway (IGW) which depends on the VPC you created. Your IGW will need the tags:

- `project_name` with the value `module02`
- `Name` with the value `module02-igw`

Create a route table that depends on the VPC and the IGW. Your route table will have to implement a route linking the `0.0.0.0/0` CIDR Block with the IGW. The `0.0.0.0/0` is really important in the IP search process, it means if you cannot find the IP you are looking for within the VPC then search it on other networks (through the IGW). Your route table will need the following tags:

- `project_name` with the value `module02`
- `Name` with the value `module02-rt`

You thought you were finished ? We now need to associate our subnets to the route table ! Create route table associations for both of your subnets. They will depend on the route table and the concerned subnet.

Exercise 11 - Autoscaling

Turn-in directory:	ex11
Files to turn in:	02_asg.tf, *.tf, *.tfvars
Forbidden function:	None
Remarks:	n/a

Any Cloud provider is based on a pay as you go system. This system allows us not to pay depending on the number of users we have. If we have 10 users our t2.micro EC2 may be sufficient for our Flask application but if tomorrow 1000000 users want to try our super API we have to find a way to scale our infrastructure!

This can be done trough autoscaling groups. The more traffic we have the more EC2 instances will spawn to handle the growing traffic. Of course those new instances will be terminated if the number of users goes down.

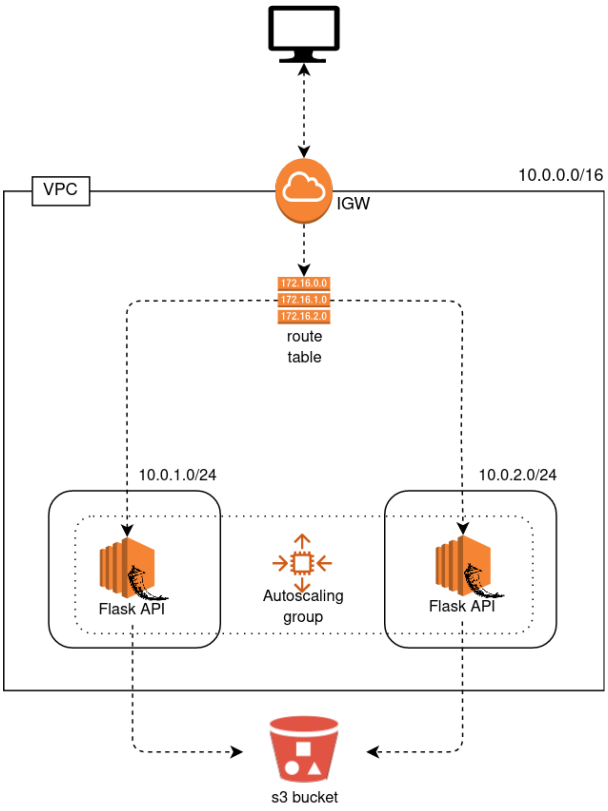

Figure 4: Flask API AWS infrastructure

Exercise

Transform your `02_ec2.t` terraform file into `02_asg.tf`. You will have to transform your code into an autoscaling group.

You need to implement a launch configuration with your EC2 parameters and add a create before destroy lifecycle.

Create an autoscaling group with :

- dependency on the launch configuration
- a link to the subnets of our vpc
- the launch configuration you previously created
- a minimal and maximal size of your autoscaling group will be 2 (this will allow us to always keep 2 instances up even if one terminates)

- a tag with:
  - `Autoscaling Flask` for a key
  - `flask-asg` for the value
  - the propagate at launch option

You should see 2 EC2 instances created within your AWS console.

Exercise 12 - Load balancer

Turn-in directory:	ex12
Files to turn in:	03_elb.tf, *.tf, *.tfvars
Forbidden function:	None
Remarks:	n/a

Let's finish our infrastructure! With our autoscaling group, we now have 2 instances but we still need to go to our AWS console to search the IP of each EC2 instance which is not convenient!

A solution is to create a load balancer. A load balancer as its name indicates will balance the traffic between EC2 instances (of our autoscaling group here).

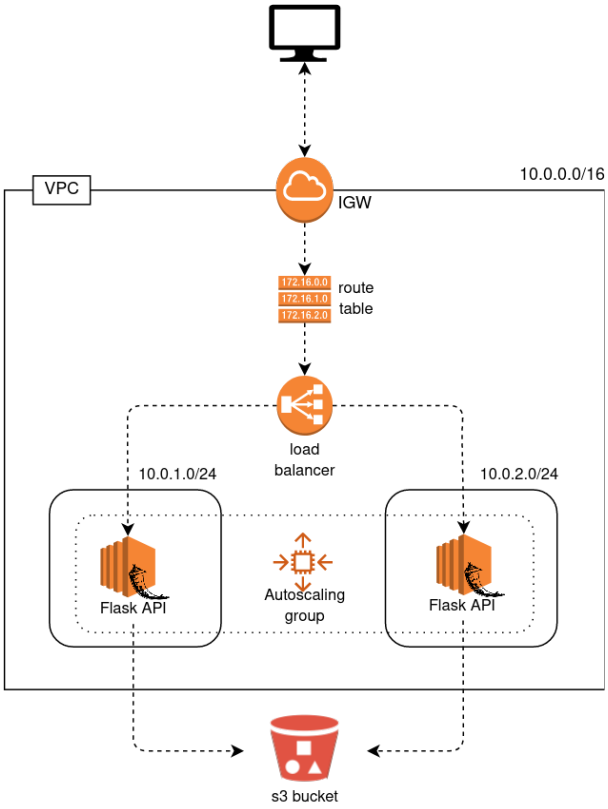

Figure 5: Flask API AWS infrastructure

Exercise

Create a security group for your load balancer. It must:

- depend on the vpc you created
- allow traffic from port 5000

Create a load balancer with:

- a health check on port 5000 every 30 sec and a healthy threshold at 2
- a listener on the port 5000
- the cross-zone load balancing option

In your autoscaling group add your load balancer and a health check type of type `ELB`.

Create a terraform output that will display the DNS name of your load balancer (this output will replace the output ip of the EC2 we had).

You should be able to use the DNS name of your load balancer to call the API now (yes this should work with the ip option you created without any other modifications)!

After the `terraform apply` finished you probably will have to wait 30 sec - 1 minute before the API is working.

Do not forget to `terraform destroy` at the end of the module!