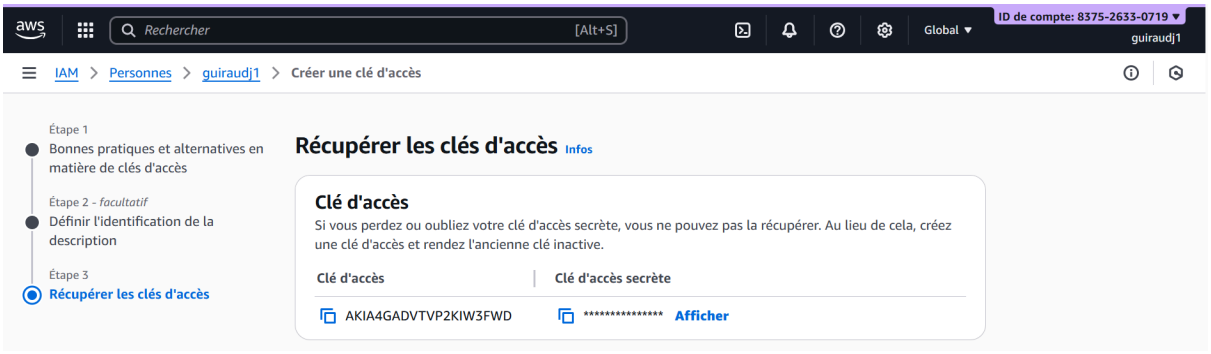
Lab 2:



Section 2: Deploying an EC2 instance using a bash script

I connected to AWS using aws configure with my access keys, selecting the region (us-east-2) and the output format (JSON).

Then I had to modify the AMI to find a valid one for Ubuntu, since AMIs depend on the selected region.

```
-------------------------------------------------------------------------------------------
|                                      DescribeImages                                     |
+-----------------------+-----------------------------------------------------------------+
|  ami-0403a1833008b227d |  ubuntu/images/hvm-ssd/ubuntu-jammy-22.04-amd64-server-20250911  |
|  ami-034bcb306215cad52 |  ubuntu/images/hvm-ssd/ubuntu-jammy-22.04-amd64-server-20250925  |
|  ami-08a98e6d1b5aa6099 |  ubuntu/images/hvm-ssd/ubuntu-jammy-22.04-amd64-server-20250801  |
|  ami-03f708ff292944dcb |  ubuntu/images/hvm-ssd/ubuntu-jammy-22.04-amd64-server-20250819  |
|  ami-085438ce84ab3ac76 |  ubuntu/images/hvm-ssd/ubuntu-jammy-22.04-amd64-server-20250617  |
|  ami-0c5ddb3560e768732 |  ubuntu/images/hvm-ssd/ubuntu-jammy-22.04-amd64-server-20251015  |
|  ami-05eb56e0befdb025f |  ubuntu/images/hvm-ssd/ubuntu-jammy-22.04-amd64-server-20250712  |
|  ami-0376da4f943e28a68 |  ubuntu/images/hvm-ssd/ubuntu-jammy-22.04-amd64-server-20250425  |
|  ami-001209a78b30e703c |  ubuntu/images/hvm-ssd/ubuntu-jammy-22.04-amd64-server-20250918  |
|  ami-08ad03b8ba2eba82f |  ubuntu/images/hvm-ssd/ubuntu-jammy-22.04-amd64-server-20250627  |
|  ami-0b05d988257befbbe |  ubuntu/images/hvm-ssd/ubuntu-jammy-22.04-amd64-server-20250516  |
|  ami-07e7b4e1174e7d3ba |  ubuntu/images/hvm-ssd/ubuntu-jammy-22.04-amd64-server-20251001  |
|  ami-0f09ef696435ff61a |  ubuntu/images/hvm-ssd/ubuntu-jammy-22.04-amd64-server-20250508  |
|  ami-0d9a665f802ae6227 |  ubuntu/images/hvm-ssd/ubuntu-jammy-22.04-amd64-server-20250822  |
+-----------------------+-----------------------------------------------------------------+
```

Then, I had to check which instance types were eligible for the Free Tier (which also depends on the region), because the one I initially chose did not work.

```
----------------------
|DescribeInstanceTypes|
+---------------------+
|  t4g.small          |
|  c7i-flex.large     |
|  t3.micro           |
|  t4g.micro          |
|  m7i-flex.large     |
|  t3.small           |
+---------------------+
```

I modified my deploy-ec2-instance.sh file with the appropriate AMI and instance type.

I get my EC2 instance:

```
(base) justine@OrdideJustine:~/devops_base/td2/scripts/bash$
./deploy-ec2-instance.sh
Instance ID = i-09c9a1c9a61949263
Security Group ID = sg-05b26674fb97eeb9e
Public IP = 3.140.251.149
```

We change the Security Group to authorize the port 8080 then we can see the “Hello World” print on http://3.140.251.149:8080


```
Hello, World!
```

Or with the command curl:

```
curl http://3.140.251.149:8080
Hello, World!
```

Exercise 1:

If I run a second time the script, an eccor occurs because the security group has already being created:

```
(base) justine@OrdideJustine:~/devops_base/td2/scripts/bash$
./deploy-ec2-instance.sh

An error occurred (InvalidGroup.Duplicate) when calling the
CreateSecurityGroup operation: The security group 'sample-app' already exists
for VPC 'vpc-0d369819d504960eb'
```

AWS uses unique names for resources like Security Groups. Attempting to create a resource with the same name results in an error because my script does not check whether the SG exists before creating it.

To delete the sample app running, we first need to delete the instance created:

```
aws ec2 terminate-instances --instance-ids i-09c9a1c9a61949263
```

→ verify that the state is at terminated

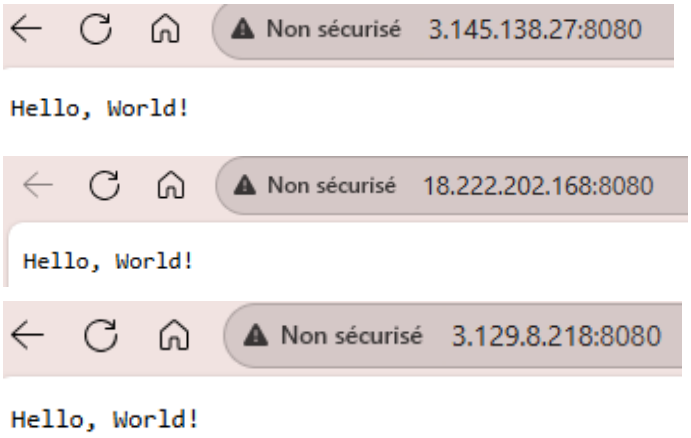
```
aws ec2 delete-security-group --group-name sample-app
```

Exercise 2:

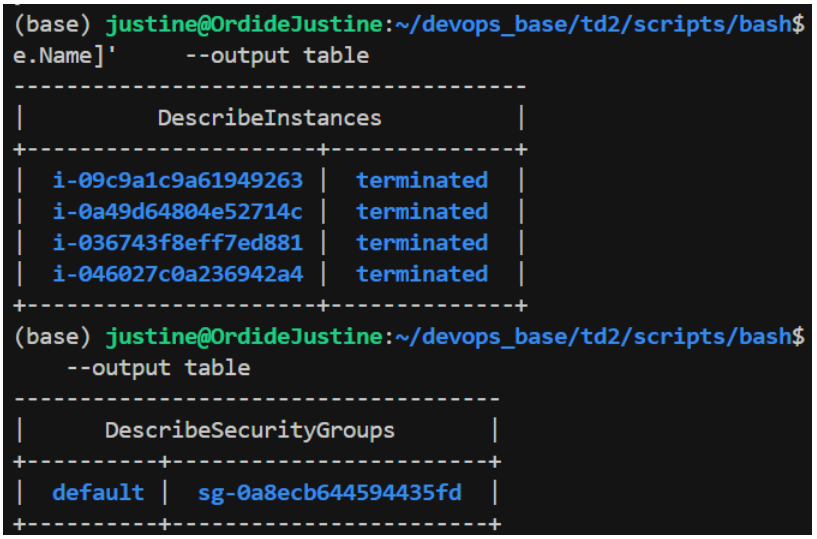
To deploy multiple instances, I replace the single run-instances command with a loop and make sure each instance has a unique tag.

I chose to make the loop run 3 times to launch multiple instances.
I give each instance a different name (sample-app-1, sample-app-2, sample-app-3) so AWS doesn’t complain about duplicate resource names.
I wait for each instance to be running before moving on.
I get the public IP of each instance so I can test my Node.js app on every server.

```
Instance 1 ID = i-036743f8eff7ed881
Instance 1 Public IP = 3.145.138.27
Instance 2 ID = i-046027c0a236942a4
Instance 2 Public IP = 18.222.202.168
Instance 3 ID = i-0a49d64804e52714c
Instance 3 Public IP = 3.129.8.218
Instance ID = i-0a49d64804e52714c
Security Group ID = sg-07300050091677a9a
Public IP = 3.129.8.218
```

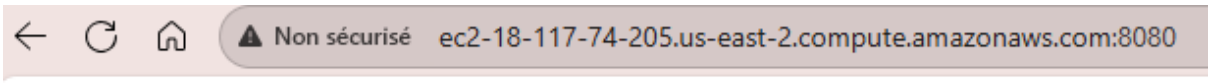


Every instance has been stopped and the security groups deleted.



Section 3: Deploying an EC2 instance using Ansible

```
(base) justine@OrdideJustine:~/devops_base/td2/scripts/ansible$ ansible -i
inventory.aws_ec2.yml _ch2_instances -m debug -a "msg={{ ansible_host }}"

ec2-18-117-74-205.us-east-2.compute.amazonaws.com | SUCCESS => {
    "msg": "ec2-18-117-74-205.us-east-2.compute.amazonaws.com"
}
```




```
(base) justine@OrdideJustine:~/devops_base/td2/scripts/ansible$ curl
http://ec2-18-117-74-205.us-east-2.compute.amazonaws.com:8080
Hello, World!
```

Exercise 3:

```
PLAY RECAP
*********************************************************************************
*****************************************************
ec2-18-117-74-205.us-east-2.compute.amazonaws.com : ok=5    changed=2
unreachable=0    failed=0    skipped=0    rescued=0    ignored=0
```

When we run the playbook a second time, most tasks make no changes because Ansible is **idempotent**: it checks the current state before applying modifications.

- **Install Node.js**: Ansible detects that Node.js is already installed, so the task reports ok without doing anything.
- **Copy sample app:** The file already exists and hasn't changed, so this task also reports ok.
- **Start sample app:** If not implemented with proper idempotency (e.g., using nohup node app.js &), this task can fail because the app is already running on port 8080 (EADDRINUSE error).

Conclusion: Ansible ensures that tasks only make changes when necessary. Out of five tasks, only two were marked as changed; the rest were already in the desired state. This demonstrates that running the playbook multiple times does not modify the system unnecessarily. Tasks that alter runtime state, like starting processes, require careful design (e.g., using PID files, systemd, or pm2) to maintain idempotency and avoid errors on repeated runs.

Exercise 4:

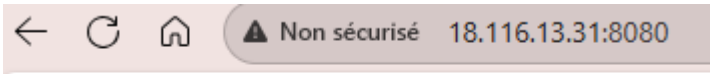
We add count=3 in *create_ec2_instance_playbook.yml* to create multiple instances (in this case, 3). We must delete our previous instances and the key that was already created, otherwise we are not able to create a new one corresponding to our 3 new instances.

```
Ansible ch2_instances
Name    sample-app-ansible
i-0054cab83971c1422     running 18.116.13.31
Ansible ch2_instances
Name    sample-app-ansible
i-0788d7205f7a1cfec     running 18.119.112.43
Name    sample-app-ansible
Ansible ch2_instances
i-0abc132eace64dec3     running 3.22.74.225
```

We add a filter in the inventory to include our created instances:

```
filters:
  "tag:Name": "sample-app-ansible"
```

```
PLAY RECAP *********************************************************************************************************************
ec2-18-116-13-31.us-east-2.compute.amazonaws.com : ok=5    changed=4    unreachable=0    failed=0    skipped=0    rescued=0    ignored=0
ec2-18-119-112-43.us-east-2.compute.amazonaws.com : ok=5    changed=4    unreachable=0    failed=0    skipped=0    rescued=0    ignored=0
ec2-3-22-74-225.us-east-2.compute.amazonaws.com : ok=5    changed=4    unreachable=0    failed=0    skipped=0    rescued=0    ignored=0
```




```
Hello, World!
```


```
Hello, World!
```


```
Hello, World!
```

Section 4: Creating a VM Image Using Packer

```
Build 'amazon-ebs.amazon_linux' finished after 5 minutes 23
seconds.
==> Builds finished. The artifacts of successful builds are:
--> amazon-ebs.amazon_linux: AMIs were created:
us-east-2: ami-0527e710054ea0fdd
```

Exercise 5:

If we run packer build a second time, Packer will create a completely new AMI. This is because the ami_name in the template usually includes a timestamp, which ensures that each AMI has a unique name. A new EC2 instance will be launched, all provisioners will run again, and a new AMI will be created. The first AMI remains unchanged, and all temporary resources (instance, keypair, security group) are recreated and cleaned up for the new build.

```
=> Wait completed after 4 minutes 33 seconds

==> Builds finished. The artifacts of successful builds are:
--> amazon-ebs.amazon_linux: AMIs were created:
us-east-2: ami-04da06141261af676
```

```
-------------------------------------------------------------------------------
|                            DescribeImages                                   |
+-------------------------+-----------------------------------------------------+
|  ami-0527e710054ea0fdd  |  sample-app-packer-b35376ab-e9b8-4eff-85db-bdb301ccc805  |
|  ami-04da06141261af676  |  sample-app-packer-97625765-55be-459a-8dfc-4ffed464b909  |
+-------------------------+-----------------------------------------------------+
```

Exercise 6:
We modified the Packer template to use the virtualbox-iso builder, which creates a local Ubuntu VM image in VirtualBox instead of an AWS AMI. The template downloads an Ubuntu ISO, provisions it with Node.js and the sample app, and shuts down the VM to produce a VirtualBox image (OVA file). This allows me to test the environment locally without relying on a cloud provider.

Section 5: Deploying, Updating, and Destroying an EC2 Instance Using OpenTofu

```
Apply complete! Resources: 3 added, 0 changed, 0 destroyed.
Outputs:

instance_id = "i-07c76b6456b4cf706"
public_ip = "3.22.101.45"
security_group_id = "sg-0f21f6984ecc08894"
```

Hello, World!

When we change the tags by adding update, it does not recreate an instance but only updates it (so I have the same outputs):

```
~ update in-place

OpenTofu will perform the following actions:

  # aws_instance.sample_app will be updated in-place
  ~ resource "aws_instance" "sample_app" {
        id                                   = "i-07c76b6456b4cf706"
      ~ tags                                 = {
            "Name" = "sample-app-tofu"
          + "Test" = "update"
        }
      ~ tags_all                             = {
          + "Test" = "update"
            # (1 unchanged element hidden)
        }
        # (31 unchanged attributes hidden)
        # (9 unchanged blocks hidden)
    }

Plan: 0 to add, 1 to change, 0 to destroy.


Apply complete! Resources: 0 added, 1 changed, 0 destroyed.

Outputs:
instance_id = "i-07c76b6456b4cf706"
public_ip = "3.22.101.45"
security_group_id = "sg-0f21f6984ecc08894"
```

tofu destroy

```
Destroy complete! Resources: 3 destroyed.
```

Exercise 7:

If tofu apply is run after the resources have been destroyed, OpenTofu detects that the defined resources no longer exist in the cloud environment. As a result, it recreates all the resources specified in the configuration, including the EC2 instance, security groups, and any tags. The behavior is equivalent to applying the configuration for the first time. The new resources will have different IDs, but their final state and configuration will match what was originally defined in the Terraform files. This demonstrates the declarative nature of OpenTofu: it ensures that the actual infrastructure always matches the desired state described in the configuration.

```
instance_id = "i-0bee285c05676c715"
public_ip = "18.116.204.54"
security_group_id = "sg-0ccae9a531b167bc8"
```

Exercise 8:

To deploy multiple EC2 instances, we modified main.tf by adding the count argument (count=3) to the aws_instance resource, so it will create three instances using the same AMI (and security group).
We also updated outputs.tf to account for the multiple instances. Since the resource now uses count, its attributes must be accessed with an index.

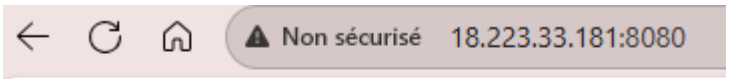
```
Apply complete! Resources: 2 added, 0 changed, 0 destroyed.

Outputs:

instance_ids = [
  "i-0bee285c05676c715",
  "i-0be1b560200c69876",
  "i-005d4159b86386fc7",
]
public_ips = [
  "18.116.204.54",
  "18.221.219.57",
  "3.142.68.244",
]
security_group_id = "sg-0ccae9a531b167bc8"
```

Section 6: Deploying an EC2 Instance Using an OpenTofu Module

We used the instance ami-0b6c1518053a48192 created with Packer.

```
i-0b20ef583a37d060c |  18.220.242.88
i-0ac94cf64d13f4a7c |  18.223.33.181
Apply complete! Resources: 6 added, 0 changed, 0 destroyed.
```

Hello, World!



Hello, World!

Exercise 9:

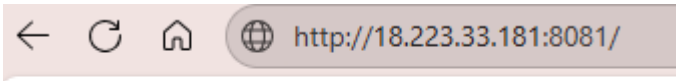
We modified the EC2 module to accept an additional input variable port and updated the user_data script to start the application on the specified port. Then, in the root module, we passed different port values (8081 and 8082) to each instance so that they run on separate ports. This allows each EC2 instance to listen on its assigned port while reusing the same module.

```
module "sample_app_1" {
  source = "../../../modules/ec2-instance"
  ami_id = "ami-0b6c1518053a48192"  # Replace with your actual AMI ID
  name   = "sample-app-tofu-1"
  port   = 8081
}

module "sample_app_2" {
  source = "../../../modules/ec2-instance"
  ami_id = "ami-0b6c1518053a48192"  # Replace with your actual AMI ID
  name   = "sample-app-tofu-2"
  port   = 8082
}
```

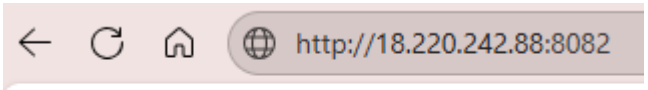
When we check the security groups:


```
"SecurityGroups": [
        {
            "GroupId": "sg-09d7db3acce13143d",
            "IpPermissionsEgress": [],
            "VpcId": "vpc-0d369819d504960eb",
            "SecurityGroupArn":
"arn:aws:ec2:us-east-2:837526330719:security-group/sg-09d7db3acce13143d",
            "OwnerId": "837526330719",
            "GroupName": "sample-app-tofu-1",
            "Description": "Allow HTTP traffic into the sample app",
            "IpPermissions": [
                {
                    "IpProtocol": "tcp",
                    "FromPort": 8081,
                    "ToPort": 8081,
                    "UserIdGroupPairs": [],
                    "IpRanges": [
                        {
                            "CidrIp": "0.0.0.0/0"
                        }
                    ],
```

```
                    "Ipv6Ranges": [],
                    "PrefixListIds": []
                }
            ]
        },
        {
            "GroupId": "sg-076a27cd75fdd6ac4",
            "IpPermissionsEgress": [],
            "VpcId": "vpc-0d369819d504960eb",
            "SecurityGroupArn":
"arn:aws:ec2:us-east-2:837526330719:security-group/sg-076a27cd75fdd6ac4",
            "OwnerId": "837526330719",
            "GroupName": "sample-app-tofu-2",
            "Description": "Allow HTTP traffic into the sample app",
            "IpPermissions": [
                {
                    "IpProtocol": "tcp",
                    "FromPort": 8082,
                    "ToPort": 8082,
                    "UserIdGroupPairs": [],
                    "IpRanges": [
                        {
                            "CidrIp": "0.0.0.0/0"
                        }
                    ],
                    "Ipv6Ranges": [],
                    "PrefixListIds": []
                }
            ]
        }
    ]
}
```

Hello, World!



Hello, World!

Exercise 10:

We modified the module by adding the count argument in the EC2 instance resource. This allows me to deploy multiple instances from a single resource block without duplicating code in the root module. I also added an instance_count variable in variables.tf so that the number of instances can be easily adjusted. Each instance gets a unique name by combining the base name with the count.index, and all other parameters, like AMI ID and port, are reused across the instances.

```
Apply complete! Resources: 6 added, 0 changed, 0 destroyed.

Outputs:
instance_ids = [
  "i-0a1b2c3d4e5f67890",
  "i-0f1e2d3c4b5a67891",
  "i-0123456789abcdef0"
]

public_ips = [
  "18.223.33.181",
  "18.220.242.88",
  "3.142.68.244"
]
security_group_id = "sg-09d7db3acce13143d"
```