A Scalable Microservices-based Web Application in a Public Cloud

Process Based Calculator

Management and Administration of IT Infrastrutures and Services

Team nr.: 12

86923: Sara Machado

90770: Rafael Figueiredo

90774: Ricardo Grade

Information Systems and Computer Engineering

IST-TAGUSPARK

2021/2022

Contents

List of Figures

Chapter 1

Summary

1.1 Introduction

This work has 3 goals:

- Deploy our *Process Based Calculator* application in the cloud, whose chosen platform was Amazon Web Services (AWS). We deploy the infrastructure with the help of Terraform and configured it with Ansible.
- Set up a monitoring system for the servers that make up the system, for this we use Prometheus to collect metrics and Grafana as a graphical interface and visualization panel.
- To automate a continuous integration (CI) / continuous deployment (CD) system, we use Jenkins to do this and add a webhook to GitHub to notify Jenkins which deploys and configures the infrastructure.

The tutorial on how to provision the infrastructure with Terraform, configure the infrastructure with Ansible, and set up a continuous integration (CI) / continuous deployment (CD) tool with Jenkins, can be found on the following YouTube URL: https://youtu.be/_YkKf6NLAHA.

1.2 Application description

The application that is being deployed is a calculator whose different operations such as addition, subtraction, multiplication and division are performed by different back-

ends.

The user interacts in a first phase with the frontend, which returns a webpage that can be used to perform different operations.
When the user makes a request, it is received by a process that plays the proxy role between the user and the backends that actually perform the operations, this process is responsible for redirecting it to the respective backend that performs the desired operation.

In more detail, the components of our application are as follows:

- A web page that the user uses to interact with the system.
- A backend that plays the proxy role explained above.
- 4 backends each are responsible for doing a math operation, one for addition, one for subtraction, one for multiplication and one for division.

In addition, each backend provides an additional endpoint */metrics* that responds with metrics collected from the application, used to monitor servers. This endpoint is defined by the *prometheus-api-metrics* module.

The backends were implemented with *node express* and for package management *npm* was chosen.

1.3 Infrastructure description

The system infrastructure is made up of a diverse set of components, each with a different set of responsibilities.

The basic functionality of the system is provided by a set of 4 processes, encapsulated in a private subnet, each of which performs a different mathematical operation: addition, subtraction, multiplication and division.

To access these functionalities, it was necessary to create an entry point for customers to access, for that, a public subnet was created, which has direct access to the private subnet and is composed of the following entities: *frontend*, *monitor*, and a NAT gateway.

The *frontend* contains the page that the application's clients would access to interact with the system.

The *monitor*, from time to time, collects information, known as metrics, about the services/operations provided by the application and plots a series of graphs that visually represent the progression of the system's state. These collected metrics can also provide information to update the infrastructure as per apparent needs.

Last but not least, there is also a NAT gateway, in the public subnet, which allows processes present in the private subnet to have access to Internet contents for packages and framework updates.

Both public and private subnets are also encapsulated in a VPC which is the section of the cloud network where this system would be present. To allow this section of the cloud to have traffic, an Internet gateway is present as an entry point to this VPC.

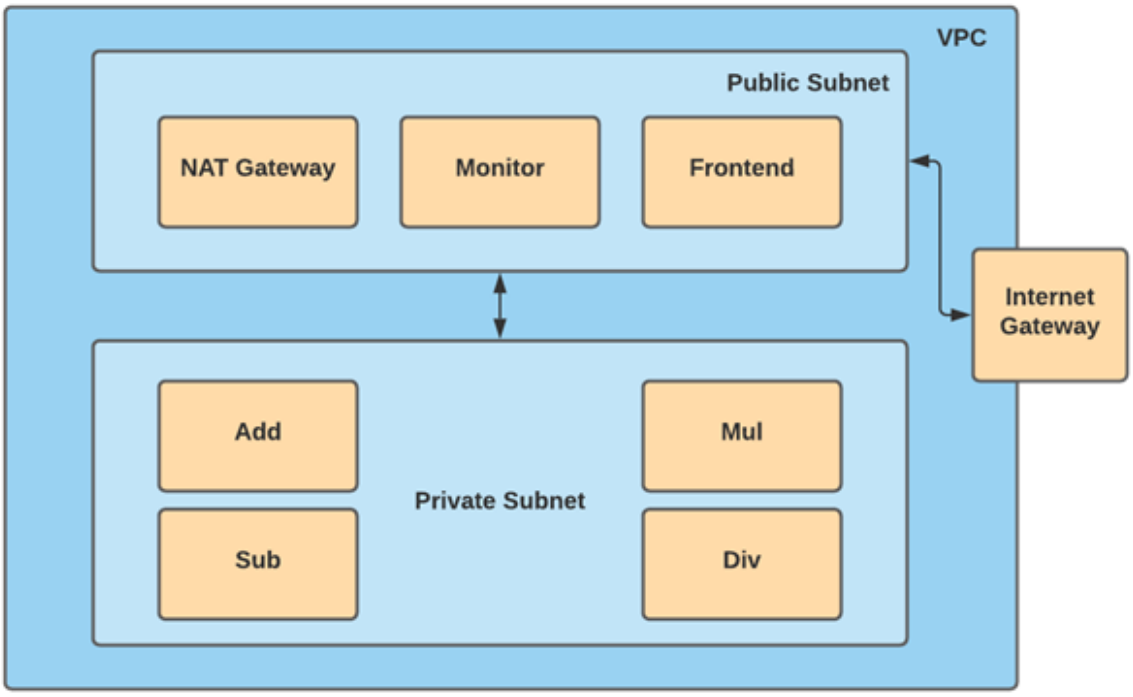

Figure 1.1: System Architecture

In the diagram (Figure 1.1) of that infrastructure, there is a visual representation of the processes present in the system and represented by the arrows, the communication that takes place between the multiple components. The overlapping of several boxes represents the encapsulation of the above mentioned components.

Chapter 2

Solution

2.1 Infrastructure provisioning using Terraform

Infrastructure provisioning is done through Terraform, where the various infrastructure components have been split into multiple files for better organization and management.

The *terraform-variables.tf* file, as the name implies, contains four variables that are used in other Terraform files and contain information to be provided to the could provider to launch the instances. Within the file, the region and the availability zone of the machines are defined, along with the type of instance and the AMI to be executed.

The *terraform-providers.tf* file defines which cloud provider we will use to build the infrastructure, in our case we chose the AWS cloud provider. Along with the provider, the path to the file containing the credentials associated with the account created with the provider is defined.

The *terraform-instances.tf* file instantiates the different types of instances that the system has along with the path to the key pair that will be needed to access each instance. Considering the system described in the previous sections, it is enough to define three types of instances: frontend, servers (operation processes) and monitor. For each instance type, the following attributes are defined:

- The AMI.
- Count, that is, the number of instances of this type that must be launched.
- The private IP.
- The subnet where the instance would be inserted.

- The name of the key pair used to access the instance.
- The set of security groups.

Finally, the file *terraform-networks.tf* contains all the information about the network on which the application will run. The following resources are defined in the file:

1. The VPC.
2. The private subnet.
3. The public subnet.
4. An internet gateway.
5. The route table associated with the internet gateway.
6. An association between the route table in (5) and the public subnet in (3).
7. A NAT gateway.
8. The NAT public IP.
9. The route table associated with the NAT gateway.
10. An association between the route table in (9) and the private subnet in (2).
11. A common security group for all instances, enabling SSH and ICMP protocol.
12. The security group specifically for gateways, allowing both HTTPS and HTTP.
13. The security group specifically for services, allowing NodeJS connections.

2.2 Infrastructure configuration using Ansible

To configure the infrastructure, Ansible was used.

First, we create an inventory file called *hosts*, which defines groups of known hosts.

We defined six hosts: *lb*, *monitor*, *addServer*, *subServer*, *mulServer* and *divServer*; along with their IPs, usernames and connection types.

We also created four groups:

- *front*: which is made up of the *lb*.
- *monit*: which is composed by the *monitor*.
- *targets*: which is made up of the *lb* and the 4 servers.
- *gatewayed*, which is made up of the servers, and uses *gatewayed:vars* to make all ssh connections through the *monitor*'s ssh tunnel to the server. Otherwise the servers would not be accessible as they only have one network identifier within their private network.

Second, we create an Ansible file where the following playbooks are defined:

- On *targets*:
  1. Updates the system
  2. Installs *nginx*, *nodejs*, *npm* and *pm2*.
  3. Loads the required server source code on each host.
  4. Installs the necessary modules through *npm*.
  5. Starts the server process on each host.
- On *front*:
  1. Deploys the web page.
- On *monit* (For that, we first need to install Grafana and Prometheus using Ansible Galaxy):
  1. Installs Grafana.
  2. Installs Prometheus and targets servers that perform math operations on port 8090.

As Terraform changes the IPs, we also create a shell script that automatically gets the IPs from the infrastructure and updates the host file, as well as the frontend web page for him to be able to contact the *lb* (which in turn plays the proxy role).

2.3 Monitoring configuration

Although we have configured Prometheus to get metrics from the servers, we still need to do a manual configuration of the dashboard:

1. Log into Grafana, which is deployed to the *monitor* on port 3000 with the username *admin* and the password *admin*.

2. Configure Prometheus as a data source, which is running on port 8090 on the math operations servers.

3. Importing a dashboard that we customize (*dashboard.json*).

Figure 2.1: Grafana Dashboard

After that, Grafana is successfully configured (Figure 2.1).

2.4 Jenkins integration

Continuous integration (CI) / continuous deployment (CD) was established using Jenkins. To do this, we integrate our git project with Jenkins to run tests and deploy the project every time we do *git push*. If the build fails, that is, an error has occurred, the infrastructure is destroyed and an email is sent to notify the person responsible for the failure.

Jenkins integration was done with the following steps:

- Install Jenkins using *https://updates.jenkins.io/latest/jenkins.war*.
- Create an SSH RSA key and add the public key to GitHub.
- Run *java -jar jenkins.war*.
- Set up an account and install all suggested plugins.
- Go to Manage Jenkins – Configure System – Available: Search for *PostBuild-Script* plugin and install it.
- Create a FreeStyle project called calculator and configure it as follows:
  - General: Select the GitHub project and provide the URL of the GitHub repository.
  - Source Code Management: Select git and proceed with the following steps:
    - Repository URL: Enter the SSH URL of the git repository.
    - Credentials: Click Add Jenkins and add the following:
      - Kind: SSH username with private key.
      - ID: Jenkins-Calculator
      - Private Key: Add the private key associated with the SSH public key that was added to GitHub earlier.
    - Branch Specifier: Change to **/main*.
  - Build Triggers: Set the GitHub hook trigger to the GITScm polling, which triggers the auto build when a *git push* occurs.
  - In Build: Add Build Step – Select the shell script option and type the command *sh DevOps/jenkinsBuild.sh*, which deploys, configures and tests the project.
  - Post-Build Actions: Click add post-build action and add the following:
    - Post-Build Task:
      - Post-Build Step – Add Build Step – Select the shell script option and type the command *sh DevOps/jenkinsFail.sh*, which destroys the infrastructure.
      - Select if build was: Failure

    - Email-Notification:
        - Recipients: E-mails from responsible persons separated by space.

- Configure Webhook:
    - To set up a webhook, for GitHub to notify Jenkins, we need to have a public IP. For this purpose *socketxp* can be used to set it up listening on *http://localhost:8080/*, this IP is hereafter called *[Jenkins-IP]*.
    - Manage Settings – Configure System – GitHub – Advanced – Specify another hook URL for GitHub configuration: *http://[Jenkins-IP]:8080/github-webhook/*.
    - GitHub Repository – Settings – Webhooks – Add Webhook – Payload URL: *http://[Jenkins-IP]:8080/github-webhook/* and select the push event.
- Configure Email-Notification:
    - Manage Settings – Configure System – Email-Notification: Set to preferred SMTP server.

Chapter 3

Results

3.1 Evaluation

To ensure that the system works correctly, we built a suite of tests in Jenkins that perform basic math operations to the frontend and checked if it returns a correct answer. In this case, it is assumed that the infrastructure assembly worked correctly and the system is operational. Otherwise, the infrastructure is dismantled and the person responsible is advised by email to take appropriate measures.

In addition, we also have information about errors that happen in the app available in the dashboard (Figure 2.1), so we can see when something goes wrong.

Furthermore, we can also see in (Figure 2.1) that our system's response latency is quite low, which may also have been somewhat influenced by half of the requests that return status code 304, meaning that the response was still in the cache. And as can be seen, all requests were completed successfully.

3.2 Conclusion

The 3 goals by which this work comprises, mentioned in Introduction, were successfully achieved.