RETO FINAL

BOOTCAMP

DEVOPS

Imagen que contiene Logotipo

Descripción generada automáticamente

Autor: Felipe Cendrero Díaz

Fecha:08/04/2024

Contenido

[Contexto y Problema 3](#_Toc163551872)

[Problema 3](#_Toc163551873)

[Propuesta 3](#_Toc163551874)

[Descripción de la Arquitectura del Sistema 4](#_Toc163551875)

[Balanceadores de Carga 4](#_Toc163551876)

[Terminadores SSL 4](#_Toc163551877)

[Bases de Datos 5](#_Toc163551878)

[Tecnologías de Telemetría 5](#_Toc163551879)

[Replicación Automática de Pods de Docker 5](#_Toc163551880)

[Arquitectura Cloud 7](#_Toc163551882)

[Despliegue en AWS 7](#_Toc163551883)

[Descripción del Ciclo de Vida 9](#_Toc163551884)

[Modelo de Desarrollo 9](#_Toc163551885)

[Pruebas Obligatorias 9](#_Toc163551886)

[Diferentes Tipos de Entornos 9](#_Toc163551887)

[Modelo de Ramas 9](#_Toc163551888)

[Herramientas de Automatización 9](#_Toc163551889)

[Tecnologías Utilizadas 10](#_Toc163551890)

## Contexto y Problema

En esta sección, establecemos el contexto del proyecto y delineamos el problema que estamos abordando. Es fundamental comprender el entorno en el que operamos y los desafíos que enfrentamos para desarrollar una solución efectiva. Al definir claramente el problema, podemos orientar nuestras acciones hacia su resolución de manera efectiva y eficiente.

## Problema

El equipo enfrenta el desafío de desarrollar una aplicación moderna y competente que pueda manejar de manera eficiente cargas variables de trabajo, garantizando una alta disponibilidad y escalabilidad.

## Propuesta

Nuestra propuesta consiste en implementar una arquitectura en la nube utilizando tecnologías de AWS, Docker y Kubernetes. Esto nos permitirá automatizar la replicación de pods de Docker para manejar cargas excesivas y desplegar entornos de desarrollo y producción en AWS con la gestión centralizada de DNS a través de Amazon Route 53.

## Descripción de la Arquitectura del Sistema

En esta sección, proporcionamos una visión general de la arquitectura de nuestro sistema. Detallamos los componentes clave, las interacciones entre ellos y cómo se combinan para ofrecer la funcionalidad deseada. Una descripción clara de la arquitectura nos ayuda a comprender la estructura subyacente de nuestra solución y nos guía en el diseño, desarrollo e implementación de la misma.

### Balanceadores de Carga

Para implementar los balanceadores de carga, se utilizará AWS Elastic Load Balancing (ELB) en su configuración de balanceo de carga de capa 7. Esto garantiza una distribución equitativa del tráfico entrante a través de múltiples destinos, mejorando así la escalabilidad y la disponibilidad de la aplicación, al utilizar los servidores eks, hemos eliminado el ELB porque ya nos lo ofrecía el cluster de kubernetes.

### Terminadores SSL

En un principio se pensó en utilizar terminadores SSL para que la aplicación fuera en https:// pero al intentar implementarlo nos suponía unos costes, por lo que descartamos el realizarlo, probamos a utilizar un certificado autofirmado pero nos dio un error al importarlo, por lo tanto se descarto este paso.

### Bases de Datos

Para las bases de datos, se utilizará Amazon RDS para implementar bases de datos relacionales como PostgreSQL o MySQL. Esto proporciona una gestión fácil y escalable de las bases de datos, con capacidades de copia de seguridad automática y escalado vertical y horizontal según sea necesario, esto se tenia pensado para hacerlo en producción, para el entorno de desarrollo, se pensó en una base de datos postgres que se levanta en con Docker-compose

### Tecnologías de Telemetría

Se propone el uso de AWS CloudWatch para la monitorización y alertas en tiempo real, así como la integración con herramientas de análisis como Prometheus y Grafana. Esto permite supervisar el rendimiento de la aplicación, detectar problemas de manera proactiva y realizar análisis detallados sobre el comportamiento del sistema.

### Replicación Automática de Pods de Docker

Se implementará la replicación automática de pods de Docker utilizando Kubernetes como orquestador de contenedores. Kubernetes proporciona capacidades de autoescalado horizontal, lo que significa que los pods adicionales se crearán automáticamente en respuesta a una carga excesiva, garantizando así un rendimiento óptimo de la aplicación en todo momento.

# Diagrama Descripción generada automáticamente

# Arquitectura Cloud

Aquí nos adentramos en la infraestructura en la nube que utilizaremos para implementar nuestra solución. Exploramos los servicios y tecnologías de la nube que aprovecharemos, así como la relación entre ellos. Una comprensión sólida de la arquitectura en la nube nos permite aprovechar al máximo los recursos disponibles y garantizar un despliegue eficiente y escalable de nuestra aplicación.

### Despliegue en AWS

**Amazon VPC (Virtual Private Cloud):** Configuraremos una VPC para nuestra infraestructura en la nube. La VPC nos permite crear una red virtual aislada en AWS donde podemos lanzar recursos de manera segura. Utilizaremos subredes públicas y privadas para separar las cargas de trabajo y aplicar controles de acceso adecuados.

**Amazon Route 53:** Utilizaremos Route 53 como servicio de DNS para asignar nombres de dominio a los entornos de desarrollo y producción en AWS. Esto facilitará el acceso a la aplicación y permitirá la gestión centralizada de la resolución de nombres en la nube.

**Amazon RDS (Relational Database Service):** Implementaremos una base de datos relacional utilizando RDS para almacenar y gestionar los datos de nuestra aplicación. RDS ofrece opciones para varios motores de bases de datos, como PostgreSQL, MySQL, MariaDB, Oracle y SQL Server, con capacidades de escalabilidad automática, copias de seguridad automáticas y alta disponibilidad.

**Amazon Elastic Load Balancing (ELB):** Configuraremos un balanceador de carga utilizando ELB para distribuir el tráfico de manera equitativa entre las instancias de EC2 que ejecutan nuestra aplicación. ELB proporciona alta disponibilidad, escalabilidad automática y monitoreo integrado para garantizar un rendimiento óptimo de la aplicación.

Diagrama

Descripción generada automáticamente

## Descripción del Ciclo de Vida

En esta sección, delineamos el ciclo de vida del software en nuestro proyecto. Detallamos el modelo de desarrollo que seguiremos, los tipos de pruebas que realizaremos, los diferentes entornos que mantendremos y el modelo de ramas que emplearemos para trabajar de manera colaborativa.

### Modelo de Desarrollo

Se seguirá un modelo de desarrollo ágil, fomentando la iteración continua y la entrega incremental del software. La aplicación estará desarrollada en Python, aprovechando su versatilidad y amplia gama de bibliotecas y frameworks disponibles, lo que hemos hecho para el entorno de desarrollo

### Pruebas Obligatorias

Se establecerán pruebas unitarias, de integración y de aceptación como parte del proceso de desarrollo. Esto garantiza la calidad del software y la detección temprana de posibles problemas, minimizando así el riesgo de errores en producción.

### Diferentes Tipos de Entornos

Se mantendrán entornos de desarrollo, pruebas y producción para diferentes fases del ciclo de vida del software. Esto permite probar cambios de manera segura antes de desplegarlos en producción, asegurando una experiencia fluida para los usuarios finales.

### Modelo de Ramas

Se empleará GitFlow como modelo de ramas Git para trabajar de manera colaborativa y gestionar el flujo de cambios de manera eficiente y ordenada. Esto facilita la colaboración entre miembros del equipo y proporciona un marco estructurado para el desarrollo y la entrega de nuevas funcionalidades.

### Herramientas de Automatización

Para la automatización del despliegue y la gestión de la infraestructura, se utilizarán Docker para la gestión de contenedores, Jenkins para la integración continua y Ansible para la automatización de la configuración del sistema. Estas herramientas permiten una entrega rápida y fiable del software, reduciendo el tiempo y el esfuerzo requeridos para implementar cambios en el sistema.

Con esta arquitectura y ciclo de vida del software, se establece una base sólida para el desarrollo, despliegue y operación de la aplicación, garantizando su escalabilidad, disponibilidad y calidad en todas las etapas del proceso.

# Tecnologías Utilizadas

Aquí enumeramos las tecnologías y herramientas que emplearemos en nuestro proyecto. Desde plataformas de contenedores hasta servicios en la nube, estas tecnologías constituyen la base de nuestra arquitectura de solución y nos proporcionan las herramientas necesarias para desarrollar, desplegar y operar nuestra aplicación de manera efectiva y eficiente.

Estas introducciones proporcionan un contexto inicial para cada uno de los aspectos clave de nuestro proyecto, estableciendo las bases para un entendimiento más profundo y detallado de nuestra solución. ¿Hay algo más en lo que pueda ayudarte?

Docker: Plataforma de contenedores para el desarrollo, implementación y ejecución de aplicaciones.

Docker-compose: Herramienta para definir y ejecutar aplicaciones Docker multi-contenedor.

Vagrant: Herramienta para la creación y configuración de entornos de desarrollo virtuales.

Jenkins: Herramienta de integración continua y entrega continua (CI/CD) para automatizar el proceso de construcción, pruebas y despliegue de software.

Terraform: Herramienta de infraestructura como código (IaC) para definir y gestionar la infraestructura de manera programática.

Git: Sistema de control de versiones distribuido para el seguimiento de cambios en el código fuente y la colaboración en equipos de desarrollo.

Herramientas Utilizadas:

AWS (Amazon Web Services):

Amazon RDS: Servicio de bases de datos relacionales en la nube.

Amazon VPC: Servicio de configuración de redes privadas virtuales en la nube.

AWS CloudWatch: Servicio de monitorización y alertas en tiempo real.

Amazon Route 53: Servicio de DNS para asignar nombres de dominio.

Draw.io: Herramienta para diagramar la arquitectura de soluciones en la nube y visualizar la interacción entre los servicios de AWS.

## Creación de un entorno local de desarrollo

Para el entorno de desarrollo, hemos levantado una infraestructura con vagrant en el que se instalan las siguientes herramientas:

Docker

Docker-compose

Git

Jenkins

El archivo en cuestión es el vagrantfile

Texto

Descripción generada automáticamente

Luego para no tener que andar complicando el despliegue hemos dockerizado la app y la hemos subido a dockerhub

Texto

Descripción generada automáticamente

Y por último para el despliegue hemos realizado un Docker-compose

Texto

Descripción generada automáticamente

Por lo que para reproducir el entorno de producción deberemos de realizar los siguientes pasos.

Clonar el siguiente repositorio:

Git clone <https://github.com/felipealbarregas/Qualentum_Reto_final.git>

Instalar o tener instalado vagrant e irnos a la ruta y ejecutar el código

Vagrant up

Conectarnos por ssh a la maquina y clonar el repositorio de nuevo y ejecutar

docker-compose up --build -d

Por último comprobamos si funciona yo he utilizado postman

Interfaz de usuario gráfica, Texto, Aplicación, Correo electrónico

Descripción generada automáticamente

## Creación de pipeline de CI.

Para la creación del pipeline, se ha configurado las credenciales de github, nos hemos encontrado problemas a la hora de realizar las conexiones con github debido a que primero se deben de configurar las conexiones ssh en la máquina, una vez resuelto esto, hemos configurado un pipeline llamado CI, que nos obtiene el repositorio de la rama developers, lo clona en local, monta los servidores Docker, una vez que ha montado los servidores Docker, ejecutamos una pruebas para confirmar que se encuentra todo correcto, y una vez que ya se ha comprobado realiza un merge con la rama MAIN.

pipeline {

agent any

triggers {

pollSCM('\* \* \* \* \*') // Esto verifica los cambios en el repositorio cada minuto

}

stages {

stage('Clonar repositorio') {

steps {

git branch: 'developers',

credentialsId: '3a4040da-536b-45ed-aa28-d249a34f1284',

url: 'git@github.com:felipealbarregas/Qualentum\_Reto\_final.git'

}

}

stage('Montar servidor') {

steps {

script {

// Establecer las variables de entorno

env.FLASK\_ENV = 'development'

env.DATABASE\_URI = 'postgresql://myuser:mypassword@database:5432/mydatabase'

// Cambiar al directorio del repositorio clonado

dir('/var/lib/jenkins/workspace/ci') {

// Ejecutar docker-compose

sh 'docker-compose up --build -d'

}

}

}

}

stage('Ejecutar pruebas') {

steps {

script {

// Cambiar al directorio del repositorio clonado

dir('/var/lib/jenkins/workspace/ci') {

// Ejecutar las pruebas unitarias

def testOutput = sh(script: 'python3 test\_api.py', returnStdout: true).trim()

// Verificar si hay errores en la salida de los tests

def testPassed = !testOutput.contains('FAILED')

if (testPassed) {

echo 'Los tests se completaron correctamente. Realizando merge a la rama main.'

script {

// Cambio al entorno remoto y operaciones de Git

sh 'git fetch --all'

sh 'git checkout main'

sh 'git pull origin main'

sh 'git merge origin/developers --no-ff -m "Merge de developers a main"'

sh 'git push origin main'

}

} else {

error 'Los tests fallaron. No se realizará el merge.'

}

}

}

}

}

}

}

Por otro lado hemos configurado otro pipeline, que no se ejecuta automáticamente al realizar cambios ya que es el CD y lo lanzamos manualmente, lo que hacemos es clonar el repositorio main, cambiarnos a la carpeta terraform, y ejecutamos terraform init y terraform aply

pipeline {

agent any

environment {

GIT\_CREDENTIALS\_ID = '3a4040da-536b-45ed-aa28-d249a34f1284'

}

stages {

stage('Clonar Repositorio') {

steps {

checkout scm: [

$class: 'GitSCM',

branches: [[name: '\*/main']],

extensions: [],

userRemoteConfigs: [[

credentialsId: '${GIT\_CREDENTIALS\_ID}',

url: 'git@github.com:felipealbarregas/Qualentum\_Reto\_final.git'

]]

]

}

}

stage('Aplicar Terraform') {

steps {

script {

dir('/var/lib/jenkins/workspace/CD/terraform/') {

withCredentials([[

$class: 'AmazonWebServicesCredentialsBinding',

credentialsId: 'c7dfe327-bee8-4f22-99e2-600c3a895409',

accessKeyVariable: 'AWS\_ACCESS\_KEY\_ID',

secretKeyVariable: 'AWS\_SECRET\_ACCESS\_KEY'

]]) {

sh 'terraform init'

sh 'terraform apply -auto-approve'

}

}

}

}

}

}

}

## Infraestructura como código.

Hemos configurado con terraform la infraestructura que vamos a levantar en aws, hemos decidido configurarlo todo en el main.tf ya que nos es mucho mas sencillo a la hora de

Dejamos la estructura en la carpeta del repositorio terraform/

terraform {

  required\_version = ">= 0.13"

  required\_providers {

    kubectl = {

      source  = "gavinbunney/kubectl"

      version = ">= 1.10.0"

    }

  }

}

provider "aws" {

  region = "eu-west-1"  # Reemplaza con tu región preferida

}

resource "aws\_vpc" "my\_vpc" {

  cidr\_block = "172.16.0.0/16"

}

resource "aws\_internet\_gateway" "gateway" {

  vpc\_id = aws\_vpc.my\_vpc.id

}

resource "aws\_subnet" "public\_subnet\_1a" {

  vpc\_id            = aws\_vpc.my\_vpc.id

  cidr\_block        = "172.16.0.0/24"

  availability\_zone = "eu-west-1a"

}

resource "aws\_subnet" "public\_subnet\_1b" {

  vpc\_id            = aws\_vpc.my\_vpc.id

  cidr\_block        = "172.16.1.0/24"

  availability\_zone = "eu-west-1b"

}

resource "aws\_subnet" "public\_subnet\_1c" {

  vpc\_id            = aws\_vpc.my\_vpc.id

  cidr\_block        = "172.16.2.0/24"

  availability\_zone = "eu-west-1c"

}

output "vpc\_id" {

  value = aws\_vpc.my\_vpc.id

}

output "public\_subnet\_ids" {

  value = [

    aws\_subnet.public\_subnet\_1a.id,

    aws\_subnet.public\_subnet\_1b.id,

    aws\_subnet.public\_subnet\_1c.id

  ]

}

resource "aws\_db\_subnet\_group" "my\_database\_subnet\_group" {

  name       = "my-database-subnet-group"

  subnet\_ids = [

    aws\_subnet.public\_subnet\_1a.id,

    aws\_subnet.public\_subnet\_1b.id,

    aws\_subnet.public\_subnet\_1c.id

  ]

}

resource "aws\_db\_instance" "my\_database" {

  identifier            = "my-database"

  engine                = "postgres"

  instance\_class        = "db.t2.small"

  allocated\_storage     = 20

  storage\_type          = "gp2"

  engine\_version        = "16.2"

  username              = "admin"

  password              = "password"

  publicly\_accessible  = false

   user\_data = data.template\_file.init\_script.rendered

  provisioner "file" {

    source      = "path/to/your/init.sql"

    destination = "/tmp/init.sql"

  }

  # Conectando la instancia al VPC y subredes existentes

  vpc\_security\_group\_ids = [aws\_security\_group.instance\_sg.id]

  db\_subnet\_group\_name  = aws\_db\_subnet\_group.my\_database\_subnet\_group.name

  # Configuración de la copia de seguridad automatizada

  backup\_retention\_period = 7

  backup\_window           = "22:00-03:00"

}

resource "aws\_security\_group" "instance\_sg" {

  vpc\_id = aws\_vpc.my\_vpc.id

  # Reglas de seguridad para permitir acceso a SSH (puerto 22) y Jenkins (puerto 8080)

  ingress {

    from\_port   = 22

    to\_port     = 22

    protocol    = "tcp"

    cidr\_blocks = ["0.0.0.0/0"]

  }

  # Regla de egress por defecto para permitir todo el tráfico saliente

  egress {

    from\_port   = 0

    to\_port     = 0

    protocol    = "-1"

    cidr\_blocks = ["0.0.0.0/0"]

  }

  tags = {

    Name = "Instance-Security-Group"

  }

}

module "eks\_cluster\_produccion" {

  source  = "terraform-aws-modules/eks/aws"

  version = "20.8.5"

  cluster\_name    = "produccion"

  vpc\_id          = aws\_vpc.my\_vpc.id

  subnet\_ids      = [

    aws\_subnet.public\_subnet\_1a.id,

    aws\_subnet.public\_subnet\_1b.id,

    aws\_subnet.public\_subnet\_1c.id

  ]

  eks\_managed\_node\_groups = {

    app = {

      name                 = "app"

      instance\_types       = ["t2.small"]

      min\_size             = 1

      max\_size             = 4

      desired\_size         = 1

      scaling\_config = {

        scale\_down = {

          enabled          = true

          evaluation\_periods = 3

          cooldown         = 300

          adjustment\_type  = "ChangeInCapacity"

          min\_adjustment\_magnitude = 1

          type             = "cpu"

          value            = 20

        }

        scale\_up = {

          enabled          = true

          cooldown         = 300

          adjustment\_type  = "ChangeInCapacity"

          min\_adjustment\_step = 1

          type             = "ChangeInCapacity"

          value            = 1

        }

      }

      # Especificar variables de entorno para el grupo de nodos

      node\_group = {

        environment\_variables = {

          DATABASE\_URL = aws\_db\_instance.my\_database.endpoint

        }

      }

    }

  }

}

resource "kubectl\_manifest" "myapp\_deployment" {

  yaml\_body = <<-EOF

apiVersion: apps/v1

kind: Deployment

metadata:

  name: myapp

spec:

  replicas: 2

  selector:

    matchLabels:

      app: myapp

  template:

    metadata:

      labels:

        app: myapp

    spec:

      containers:

      - name: myapp

        image: fecendrer/reto\_final:app

        ports:

        - containerPort: 80

---

apiVersion: v1

kind: Service

metadata:

  name: myapp-service

spec:

  selector:

    app: myapp

  ports:

    - protocol: TCP

      port: 80

      targetPort: 80

  type: LoadBalancer  # Cambiar el tipo de Service a LoadBalancer

---

apiVersion: networking.k8s.io/v1

kind: Ingress

metadata:

  name: myapp-ingress

spec:

  rules:

  - host: myapp.example.com

    http:

      paths:

      - pathType: Prefix

        path: "/"

        backend:

          service:

            name: myapp-service

            port:

              number: 80

EOF

  depends\_on = [module.eks\_cluster\_produccion]

}

resource "aws\_route53\_zone" "my\_zone" {

  name = "retofinal.qualentum"

}

resource "aws\_route53\_record" "example\_record" {

  zone\_id = aws\_route53\_zone.my\_zone.zone\_id

  name    = "app.retofinal.qualentum"

  type    = "A"

  ttl     = "300"

  records = ["example-load-balancer-dns-name"]  # Reemplaza con el DNS name de tu load balancer o el valor apropiado

}

resource "aws\_cloudwatch\_metric\_alarm" "request\_count\_alarm" {

  alarm\_name          = "Production\_Request\_Count\_Alarm"

  comparison\_operator = "GreaterThanThreshold"

  evaluation\_periods  = 1

  metric\_name         = "peticiones"

  namespace           = "MyApp/Metrics"

  period              = 60

  statistic           = "Sum"

  threshold           = 10  # Umbral para activar la alarma, ajusta según sea necesario

  alarm\_description   = "This alarm is triggered when the request count exceeds 10 in production environment."

  dimensions = {

    ApplicationName = "app"

    Environment     = "production"

  }

  alarm\_actions = ["arn:aws:sns:us-west-2:123456789012:ScaleUpTopic"]  # Acción a tomar cuando se active la alarma

}

output "db\_instance\_endpoint" {

  value = aws\_db\_instance.my\_database.endpoint

}

Con esto levantamos los servicios al ser un entorno de producción la base de datos será gestionada por el técnico y por tanto no será automatizada.