

# Documentación Obligatorio DevOps

---

En el presente documento se demuestran los avances, decisiones y observaciones referentes al Proyecto Integrador con su correspondientes referencias.

## 2.a Creación de ambientes para microservicios

---

Se utilizaron los microservicios propuestos por el Docente. Los cuatro microservicios fueron subidos a repositorios independientes y con sus correspondientes ramas: [main](#), [Develop](#) y [Testing](#).

### Enlaces a repositorios de microservicios.

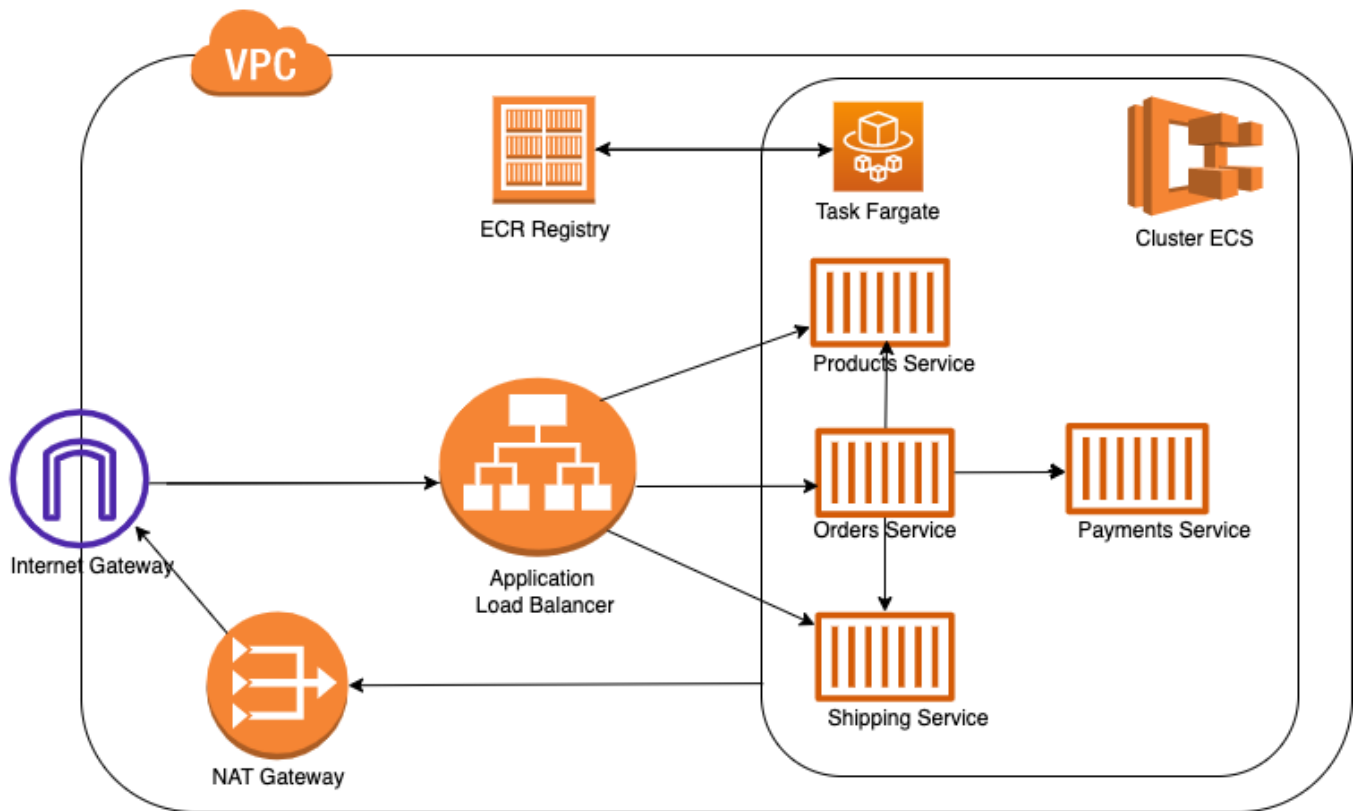
A continuación se listan los enlaces para cada repositorio de microservicio:

- [payments-service-example](#)
- [orders-service-example](#)
- [shipping-service-example](#)
- [products-service-example](#)

Como se podrá observar, ese conjunto de repositorios se encuentra en una Organización de Github creada para agrupar también la [documentación](#) y el repositorio de [infraestructura](#) o DevOps.

## 2.b Empaquetado en containers y despliegue en AWS

### Arquitectura de Servicios en AWS



En el diagrama se muestra la solución desplegada en un cluster **ECS** con **Fargate**.

Se agregó un load balancer para repartir la carga entre los microservicios.

El **ECR Registry** es utilizado para subir y disponibilizar la imagen de los contenedores de cada microservicio.

A su vez, para dar conectividad por internet, se agregó el **internet gateway** con su correspondiente **NAT gateway** y esto habilitó la comunicación a través de internet a los siguientes puertos:

- Products Service **8080**
- Payments Service **8081**
- Shipping Service **8082**
- Orders Service **8083**

### Docker Hub

Como backup, se utilizó Docker Hub para subir las imágenes de los contenedores en paralelo a AWS. Para ello fue necesario agregar este paso en **Github Actions** con el conjunto de credenciales correspondiente.

## 2.c Testeo y resultados con Postman

Cada Microservicio soporta las siguientes peticiones a través de sus endpoints:

- Products: **GET**
- Orders: **POST** (no configurado)
- Shipping: **GET** (no configurado)


Para realizar pruebas básicas, se procedió mediante la ejecución de las siguientes pruebas manuales:

- Petición **GET** a la URL **myapp-load-balancer-1298837249.us-east-1.elb.amazonaws.com** para que responda **404 Error**


The screenshot shows the Postman interface for a GET request to the URL **myapp-load-balancer-1298837249.us-east-1.elb.amazonaws.com**. The request is configured with the method **GET** and the URL. The **Headers** tab is selected, showing 6 hidden headers. The **Body** tab is also visible, showing a JSON response:

```
1 {
2   "timestamp": "2022-07-21T00:33:49.864+00:00",
3   "status": 404,
4   "error": "Not Found",
5   "path": "/"
6 }
```


- Petición **GET** a la URL **myapp-load-balancer-1298837249.us-east-1.elb.amazonaws.com/products** para que responda **200 OK** y el listado de objetos de productos hardcodedos.

GET myapp-load-balancer-  + ...

myapp-load-balancer-1298837249.us-east-1.elb.amazonaws.com/products



GET  myapp-load-balancer-1298837249.us-east-1.elb.amazonaws.com/products

Params Authorization **Headers (6)** Body Pre-request Script Tests Settings

Headers  6 hidden

KEY	VALUE
Key	Value

Body Cookies Headers (4) Test Results

Pretty Raw Preview Visualize JSON  

```
1  [
2    {
3      "id": "111",
4      "name": "Producto 111",
5      "stock": 10,
6      "description": "Este es un producto"
7    },
8    {
9      "id": "123",
10     "name": "Producto 123",
11     "stock": 10,
12     "description": "Este es un producto"
13   },
14   {
15     "id": "321",
16     "name": "Producto 321",
17     "stock": 10,
18     "description": "Este es un producto"
19   }
20 ]
```

- Petición GET a la URL `myapp-load-balancer-1298837249.us-east-1.elb.amazonaws.com/products/321` para que responda 200 OK y la información del producto con ID=321.

GET myapp-load-balancer-

+

...

myapp-load-balancer-1298837249.us-east-1.elb.amazonaws.com/products/321

GET

myapp-load-balancer-1298837249.us-east-1.elb.amazonaws.com/products/321

Params

Authorization

Headers (6)

Body

Pre-request Script

Tests

Settings

Headers

6 hidden

	KEY	VALUE
	Key	Value

Body

Cookies

Headers (4)

Test Results

Pretty

Raw

Preview

Visualize

JSON

1

{

2

"id": "321",

3

"name": "Producto 321",

4

"stock": 10,

5

"description": "Este es un producto"

6

}

## 2.d Análisis de código estático, resultados y recomendaciones.

---

### Sonarcloud

Mediante la automatización con **Github Actions** se configura el análisis de código estático con SonarCloud.

Se ajustaron de forma conveniente para que permitiera detectar facilmente su reacción a los cambios. Por ejemplo, se mostrará el ajuste del **Quality Gate** llamado **Duplicated Lines (%)** el cual analiza la cantidad de líneas de código repetidas y también el **Quality Gate** llamado **Methods should not return constants**.

Primero, se modifica el archivo

`src/main/java/uy/edu/ort/devops/ordersserviceexample/OrdersServiceExampleApplication.java` proveniente del microservicio `orders-service-example`


Concretamente, para disparar el **Quality Gate** de **Methods should not return constants** se agrega en el código la siguiente función

```
public int sonarFailure(){
    int a = 10;
    int b = 50;
    if (b<a){
        b++;
        return a;
    }
    return a;
}
```

Este método devuelve una constante lo cual genera que se dispare ese **Quality Gate**

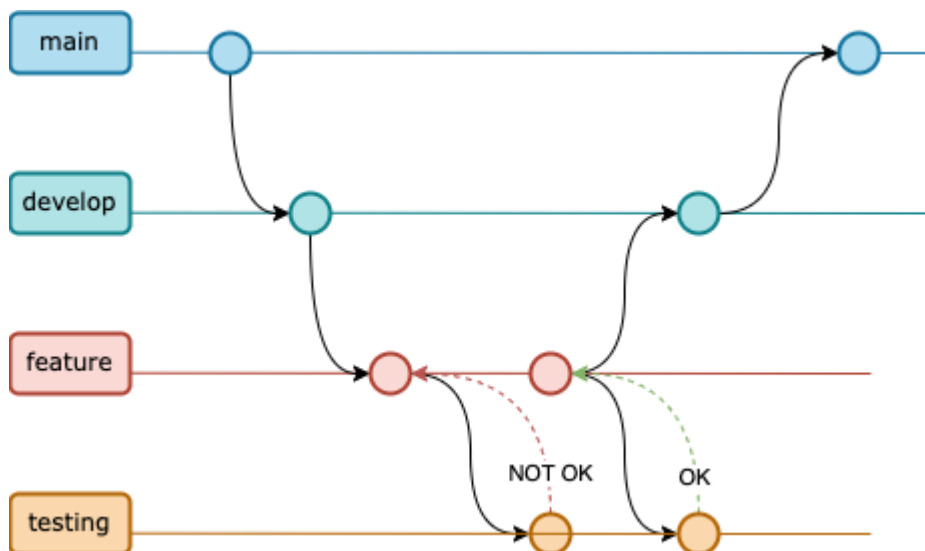
 Develop

Failed

3 minutes ago  
 ed484cec

## 2.e Utilización de Git y GitFlow para el ciclo de desarrollo.

La metodología de trabajo basada en Gitflow para el repositorio de **infraestructura** siguió las pautas según se muestra en el diagrama a continuación.

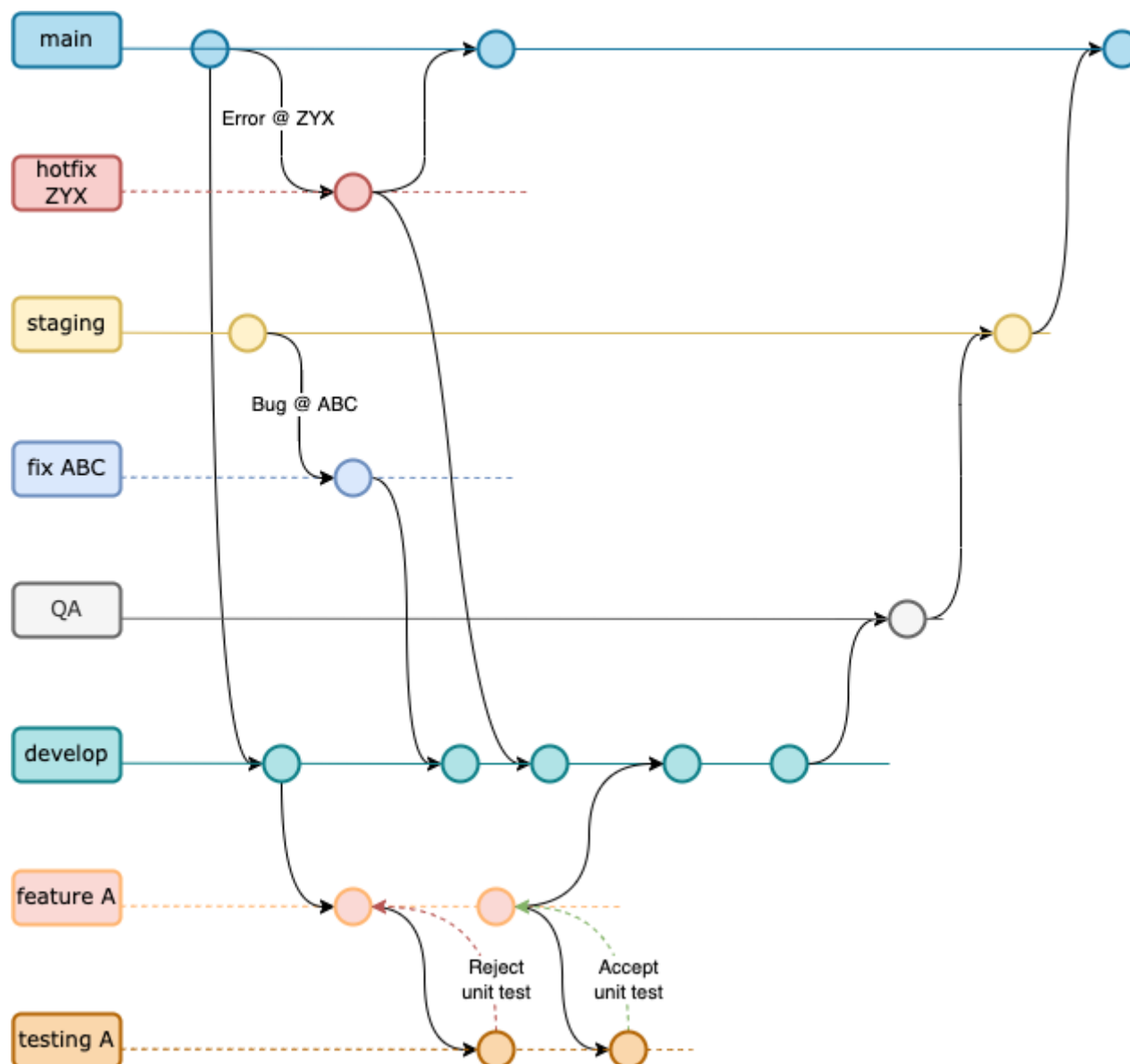


Se diseñó un proceso de trabajo básico en la cual el ambiente de producción **main**, diera partida al ambiente de **develop** y éste a su vez sirva de raíz para cada nueva feature que se desarrollara. En el diagrama se busca mostrar un caso en el que el testing es rechazado y la feature debe seguir siendo desarrollada hasta aprobada por el test unitario. Una vez logrado esto, se buscaría hacer un **merge** contra **develop**.

La realidad es que no hubo tal simultaneidad en el desarrollo de features pues se trabajó en conjunto en cada feature. Sin embargo, se buscó garantizar el trabajo de ambos participantes mediante dos formas:

- la generación de **commits** de ambos usuarios y
- la implementación de **pull requests** buscando validar y aprobar el avance entre pares.

Por otra parte, se entiende que el diagrama planteado no es del todo completo ni mucho menos infalible para un equipo de mayores dimensiones. Por lo que a continuación se planteará un diagrama basado en gitflow pero que contemple un entorno más profesional y aplicable en equipos de mayor porte.



Contemplando que se trata de una metodología de trabajo con etapas bien definidas, se agregó también un ambiente de **QA** en donde se ejecutarán pruebas sobre el sistema pero esta vez no apuntando a test unitarios sino que globales (por ejemplo, test de regresión, de integración y de rendimiento)

En la etapa de **QA** pueden detectarse problemas a solucionar los cuales entendemos es conveniente tratarlos como un **FIX** de una **feature** por lo que el equipo de desarrollo, tendrá que trabajar sobre una nueva rama como si fuera una feature nueva volviendo al ciclo inicial.

También pueden ocurrir fallos en el ambiente de **staging** en cuyo caso se entiende conveniente tratarlo de forma similar que **QA** pues no es de urgencia. Si el **FIX** es tratado independiente, y no se maneja como una feature nueva, debe volver a hacer un **merge** con **develop** para garantizar que los cambios del fix, quedan aplicados al flujo de "subida" herárquica.

Por último pero no menos imporatnte, el ambiente **main** es el ambiente de producción, y toda falla ocurrida aquí debería se corregida en un **HOTFIX** concreto y con urgencia.

La introdcucción de un **HOTFIX** debe a su vez, "bajar" al ambiente de **develop** para garantizar que en algún momento el **HOTFIX** introducido forme parte del core de desarrollo.

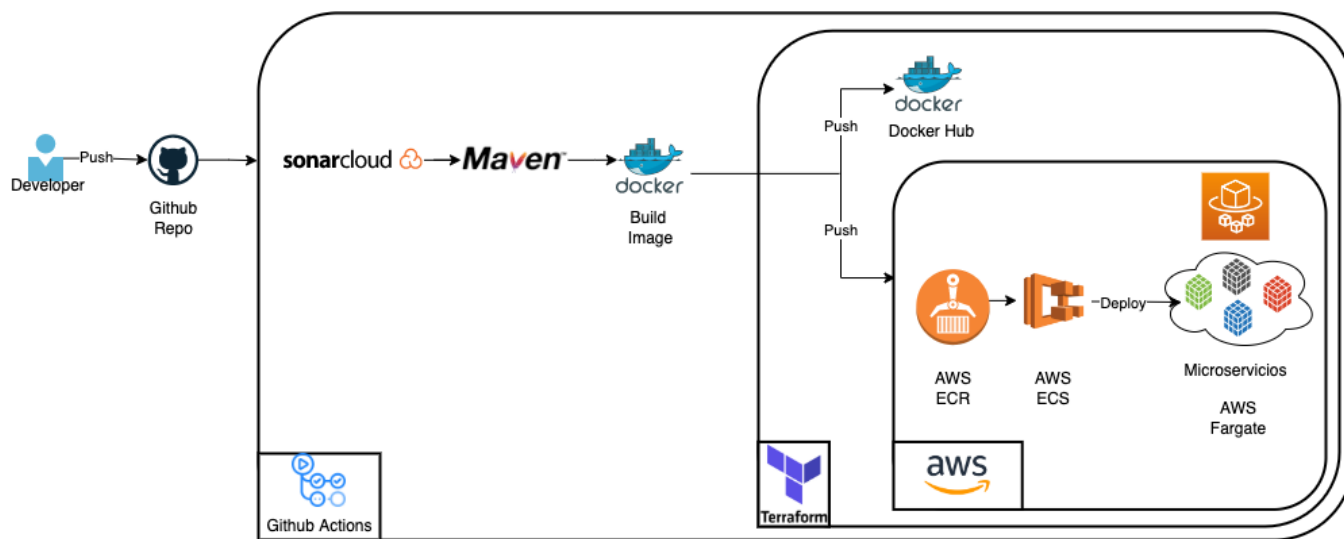


## 2.f Documentación de toda la implementación

Este paso se cumple en el presente documento.

## 2.g Diagrama CI/CD

De acuerdo a la configuración decidida por el equipo, un diagrama de Continuous Integration y Continuous Delivery podría representarse como en la siguiente imagen.



En el diagrama se muestra el proceso desde el trabajo de un desarrollador que genera un **PUSH** hacia **Github** en una rama dada (en el caso programado **testing**).

## 2.h Manejo de IaC en AWS

---

### Terraform

1. Crear archivo `main.tf`
2. Para inicializarlo correr `terraform init` en el mismo directorio del `main.tf`
3. Una vez finalizado, correr `terraform validate`
4. Una vez finalizado, correr `terraform plan`
5. Finalmente `terraform apply`

En este punto debería verse el avance de los procesos del lado de AWS.

### Bloques de código

- Networking
- AWS
- ...

### Github Actions

Se utiliza el servicio `Github Actions` para que cada `push` de `git` en la rama `Testing` dispare las acciones de CI/CD.

En cada repositorio de microservicio, se puede ver en el siguiente path `.github/workflows` lo configurado.

## 2.i Acceso al equipo docente

---

Para brindar acceso de sólo lectura al equipo docente se procede a agregar con rol `Member?` a los usuarios `ElLargo`, `mauricioamendola` y `saitama-dh` dentro de la organización `DevOps-Obligatorio` la cual contiene los repositorios y documentación necesarias para la correcta evaluación.