Documentación del Pipeline CI/CD

Este documento describe los pipelines de CI/CD para el frontend y backend.

Tabla de Contenidos

Pipeline CI/CD del Frontend

La aplicación frontend utiliza React/Vite y se despliega en AWS S3 con distribución de CloudFront.

Pipeline CI

Disparadores:

- Push a la rama `main`
- Pull requests a la rama `main`

Pasos:

1. **Descargar el repositorio**
    - Usa `actions/checkout@v4`
    - Descarga el código fuente al runner
2. **Configurar Node.js**
    - Usa `actions/setup-node@v4`
    - Instala Node.js versión 20
    - Habilita el caché de npm para builds más rápidos
    - Ruta del caché: `arquisisvitefrontend/package-lock.json`
3. **Instalar dependencias**
    - Directorio de trabajo: `./arquisisvitefrontend`
    - Ejecuta `npm ci` para una instalación limpia de dependencias
4. **Ejecutar ESLint**
    - Directorio de trabajo: `./arquisisvitefrontend`
    - Ejecuta `npm run lint` para verificaciones de calidad de código
    - **Revisa**:
        - Reglas recomendadas de ESLint (`eslint:recommended`)
        - Reglas específicas de React (`plugin:react/recommended`)
        - Variables no utilizadas (`warn`)
        - Uso de `console.log` (permitido)

        - Keys en elementos JSX (`warn`)
    - `continue-on-error: true` - no falla el pipeline si encuentra problemas de linting

5. **Construir proyecto**
    - Directorio de trabajo: `./arquisisvitefrontend`
    - Ejecuta `npm run build` para crear el build de producción
    - Entorno: `CI: false` para permitir warnings durante el build

Pipeline CD

Disparador:

- Push a la rama `main`

Ubicación: `ArquisisFrontend/.github/workflows/CD.yml`

Pasos:

1. **Descargar código**
    - Usa `actions/checkout@v4`
    - Descarga el código fuente
2. **Configurar Node.js**
    - Usa `actions/setup-node@v4`
    - Instala Node.js versión 20 con caché de npm
3. **Configurar Credenciales AWS**
    - Usa `aws-actions/configure-aws-credentials@v4`
    - Configura acceso a AWS usando secrets:
        - `AWS_ACCESS_KEY_ID`
        - `AWS_SECRET_ACCESS_KEY`
    - Región: `us-east-1`
4. **Instalar dependencias**
    - Directorio de trabajo: `./arquisisvitefrontend`
    - Ejecuta `npm ci`
5. **Construir frontend**
    - Directorio de trabajo: `./arquisisvitefrontend`
    - Ejecuta `npm run build`
    - Entorno: `CI: false`
6. **Desplegar a S3**
    - Sincroniza archivos construidos desde `arquisisvitefrontend/dist/` al bucket S3 `e1grupo27`
    - Usa la bandera `--delete` para eliminar archivos antiguos

7. **Invalidar Caché de CloudFront**
    - Crea invalidación de CloudFront para la distribución
    - Invalida todas las rutas (`/*`) para asegurar entrega de contenido actualizado

---

Pipeline CI/CD del Backend

El backend consiste en múltiples servicios Python (FastAPI, brokers, jobmaster) containerizados con Docker y desplegados en AWS EC2 usando CodeDeploy.

Pipeline CI

Triggers:

- Push a las ramas `CICD` o `main`
- Pull requests a la rama `main`

Jobs:

1. **test-api**
    - **Entorno**: Ubuntu latest, Python 3.11
    - **Directorio de trabajo**: `./api`
    - **Pasos**:
        - Descargar código
        - Configurar Python 3.11
        - Instalar dependencias desde `requirements.txt`
        - Ejecutar verificación básica de sintaxis en `main.py`, `auth.py`, `database.py`
        - Probar inicio de la API importando la aplicación FastAPI
2. **test-brokers**
    - **Estrategia**: Build matriz para `broker_requests` y `broker_updates`
    - **Entorno**: Ubuntu latest, Python 3.9
    - **Directorio de trabajo**: `./${{ matrix.service }}`
    - **Pasos**:
        - Descargar código
        - Configurar Python 3.9
        - Instalar dependencias desde `requirements.txt`
        - Ejecutar verificación de sintaxis en todos los archivos Python
3. **test-jobmaster**
    - **Estrategia**: Build matriz para `jobmaster` y `worker`
    - **Entorno**: Ubuntu latest, Python 3.11
    - **Directorio de trabajo**: `./jobmaster_service/${{ matrix.service }}`
    - **Pasos**:
        - Descargar código
        - Configurar Python 3.11

- Instalar dependencias (requirements.txt o paquetes de respaldo)
- Ejecutar verificación de sintaxis en todos los archivos Python

4. **docker-build-test**

   - **Estrategia**: Build matriz para API, broker-requests, y broker-updates
   - **Pasos**:
     - Descargar código
     - Probar build de Docker para cada servicio
     - Crea imágenes de prueba para verificar que los Dockerfiles sean válidos

Pipeline CD

No funcional, pero avanzado en mayor parte

Disparadores:

- Push a la rama `main`
- Pull requests a la rama `main`

Ubicación: `E1_arquitectura_de_software/CDfailed.yml`

Jobs:

1. deploy-backend Job (funcional)

Propósito: Construir y enviar imágenes Docker a AWS ECR Public

Pasos:

1. **Descargar código**

   - Usa `actions/checkout@v4`

2. **Configurar credenciales AWS**

   - Usa `aws-actions/configure-aws-credentials@v4`
   - Región: `us-east-1` (requerido para ECR Public)
   - Usa secrets: `AWS_ACCESS_KEY_ID`, `AWS_SECRET_ACCESS_KEY`

3. **Login a Amazon ECR Public**

   - Usa `aws-actions/amazon-ecr-login@v2`
   - Tipo de registro: público
   - Oculta password por seguridad

4. **Construir, etiquetar y push de imagen Docker para API**

   - **Directorio de trabajo**: `api/`
   - **Imagen**: `public.ecr.aws/e8f5b0r5/stocks-api:latest`
   - Construye imagen Docker y la envía a ECR Public

5. **Construir, etiquetar y push de imagen Docker para broker-updates**

- **Directorio de trabajo**: `broker_updates/`
- **Imagen**: `public.ecr.aws/e8f5b0r5/broker-updates:latest`
- Construye imagen Docker y la envía a ECR Public

6. **Construir, etiquetar y push de imagen Docker para broker-requests**

- **Directorio de trabajo**: `broker_requests/`
- **Imagen**: `public.ecr.aws/e8f5b0r5/broker-requests:latest`
- Construye imagen Docker y la envía a ECR Public

2. deploy-to-ec2 Job (con errores)

Propósito: Desplegar la aplicación en instancia EC2 usando AWS CodeDeploy

Dependencias: Requiere que el job `deploy-backend` se complete exitosamente

Pasos:

1. **Descargar código**

- Usa `actions/checkout@v4`

2. **Configurar Credenciales AWS para despliegue EC2**

- Usa `aws-actions/configure-aws-credentials@v4`
- Región: `us-east-2` (para EC2 y CodeDeploy)

3. **Debug - Listar archivos antes del empaquetado**

- Lista contenidos del directorio actual
- Verifica presencia de archivos de despliegue requeridos:
  - `docker-compose.production.yml`
  - `appspec.yml`
  - directorio `scripts/`

4. **Crear paquete de despliegue con variables de entorno**

- Crea archivo `.env` con variables del repositorio GitHub:
  - `MONGO_URI`
  - `MQTT_BROKER`
  - `MQTT_PORT`
  - `AUTH0_DOMAIN`
  - `AUTH0_AUDIENCE`
  - `URL_FRONTEND`
- Establece permisos ejecutables en scripts (`chmod +x scripts/*.sh`)
- Crea paquete ZIP de despliegue con:
  - `docker-compose.production.yml`
  - `appspec.yml`
  - directorio `scripts/`
  - archivo `.env`

5. **Subir a S3**
    - Sube `deploy.zip` al bucket S3 `fastapi-stocks-deployment-us-east-2`
6. **Crear Despliegue CodeDeploy**
    - Crea despliegue usando AWS CodeDeploy
    - **Aplicación**: `fastapi-stocks-app`
    - **Grupo de Despliegue**: `fastapi-stocks-group`
    - **Fuente**: Ubicación S3 con el archivo ZIP
    - Retorna ID de despliegue para monitoreo
7. **Esperar a que termine el despliegue (fallo permanente)**
    - Usa `aws deploy wait deployment-successful`
    - Monitorea despliegue hasta completarse o fallar

Especificación de Aplicación CodeDeploy (appspec.yml)

El proceso de despliegue está controlado por `appspec.yml` que define:

Archivos a Desplegar:

- `docker-compose.production.yml` → `/home/ubuntu/docker-compose.yml`
- `.env` → `/home/ubuntu/.env`
- `scripts/` → `/home/ubuntu/scripts/`

Hooks del Ciclo de Vida del Despliegue:

1. **BeforeInstall** (`scripts/before-install.sh`)
    - **Timeout**: 300 segundos
    - **Usuario**: root
    - **Propósito**: Instalar Docker, Docker Compose, AWS CLI, configurar usuario ubuntu
2. **ApplicationStop** (`scripts/application-stop.sh`)
    - **Timeout**: 120 segundos
    - **Usuario**: root
    - **Propósito**: Detener contenedores en ejecución, limpiar recursos
3. **AfterInstall** (`scripts/after-install.sh`)
    - **Timeout**: 300 segundos
    - **Usuario**: ubuntu
    - **Propósito**: Login a ECR, descargar últimas imágenes Docker
4. **ApplicationStart** (`scripts/application-start.sh`)
    - **Timeout**: 300 segundos
    - **Usuario**: ubuntu
    - **Propósito**: Iniciar contenedores usando docker-compose, verificar que estén ejecutándose

5. **ValidateService** (`scripts/validate-service.sh`)
    - **Timeout**: 60 segundos
    - **Usuario**: ubuntu
    - **Propósito**: Validar que la API esté respondiendo, realizar verificaciones de salud

Arquitectura de Despliegue

Frontend:

- Archivos estáticos alojados en AWS S3
- Distribuidos globalmente vía CDN CloudFront
- Invalidación automática de caché en despliegue

Backend:

- Microservicios containerizados desplegados en EC2
- Imágenes almacenadas en AWS ECR Public Registry
- Servicios orquestados con Docker Compose
- Despliegue sin tiempo de inactividad con CodeDeploy
- Los servicios incluyen:
    - API REST FastAPI
    - Broker MQTT para actualizaciones
    - Broker MQTT para solicitudes
    - Base de datos MongoDB
    - Servicio Jobmaster con workers

Variables de Entorno

El despliegue del backend usa las siguientes variables de entorno configuradas en la configuración del repositorio GitHub:

- `MONGO_URI`: Cadena de conexión MongoDB
- `MQTT_BROKER`: Hostname del broker MQTT
- `MQTT_PORT`: Puerto del broker MQTT
- `AUTH0_DOMAIN`: Dominio de autenticación Auth0
- `AUTH0_AUDIENCE`: Audiencia de la API Auth0
- `URL_FRONTEND`: URL de la aplicación frontend

Seguridad

- Credenciales AWS almacenadas como secrets de GitHub
- Credenciales de login ECR enmascaradas en logs
- Variables de entorno inyectadas de forma segura durante el despliegue
- Contenedores Docker ejecutan con permisos de usuario apropiados
- Los hooks de CodeDeploy ejecutan con privilegios mínimos requeridos