GESTOR DE EVENTOS

Por: José Miguel Dager Montoya

IMPLEMENTACIÓN DE GITLAB CI/CD

Un flujo CI/CD automatiza el proceso de desarrollo, desde la escritura del código hasta su despliegue en producción. Una buena estructuración garantizaría la entrega rápida y confiable de nuevas funcionalidades, correcciones de errores y actualizaciones. En este orden de ideas, se escoge GitLab CI/CD, ya que es una herramienta poderosa y flexible integrada en la plataforma GitLab que permite automatizar los procesos de construcción, prueba y despliegue de aplicaciones. Es una parte fundamental de los flujos de trabajo DevOps, ya que ayuda a los equipos a entregar software de manera más rápida y confiable.

En primera instancia, se configura el pipeline creando un archivo *.gitlab-ci.yml* en la raíz del proyecto; este archivo definirá los diferentes stages (etapas) del pipeline, los jobs (tareas) que se ejecutarán en cada stage y los disparadores (qué eventos desencadenan la ejecución del pipeline, como un push a una rama).

Luego se definen las siguientes etapas:

- **Build:** Compilar el código fuente del gestor de eventos.
- **Test:** Ejecutar pruebas unitarias, de integración y funcionales para asegurar la calidad del código.
- **Deploy:** Desplegar la aplicación en diferentes entornos (desarrollo, staging, producción).

Posteriormente se escriben los scripts para cada etapa, generalmente en Bash o shell, con el fin de ejecutar las tareas necesarias. Se configuran también los runners, que son aquellos agentes que ejecutan los trabajos definidos en el archivo .gitlab-ci.yml; en este caso se pueden usar runners compartidos de GitLab o configurar nuestros propios runners en la infraestructura. Por último, se personaliza el pipeline a nuestras necesidades específicas y se definen las actividades para obtener los beneficios máximos:

1. **Monitoreo y Mantenimiento:**

- Seguimiento de métricas: Monitorear el rendimiento del pipeline, como el tiempo de ejecución de cada etapa, la tasa de éxito de las pruebas y la frecuencia de los despliegues.
- Detección de anomalías: Implementación de alertas para notificar fallos o retrasos en el pipeline.
- Mantenimiento regular: Actualizar las imágenes de Docker, las herramientas y las configuraciones del pipeline para mantenerlo actualizado y seguro.

2. **Integración Continua:**

- Pruebas exhaustivas: Cubrir diferentes escenarios y mejorar la calidad del código.
- Análisis de código estático: Mediante el uso de herramientas como SonarQube para identificar problemas de código antes de que se produzcan errores en producción.
- Code review: Fomentar la revisión de código para garantizar la calidad y coherencia del código.

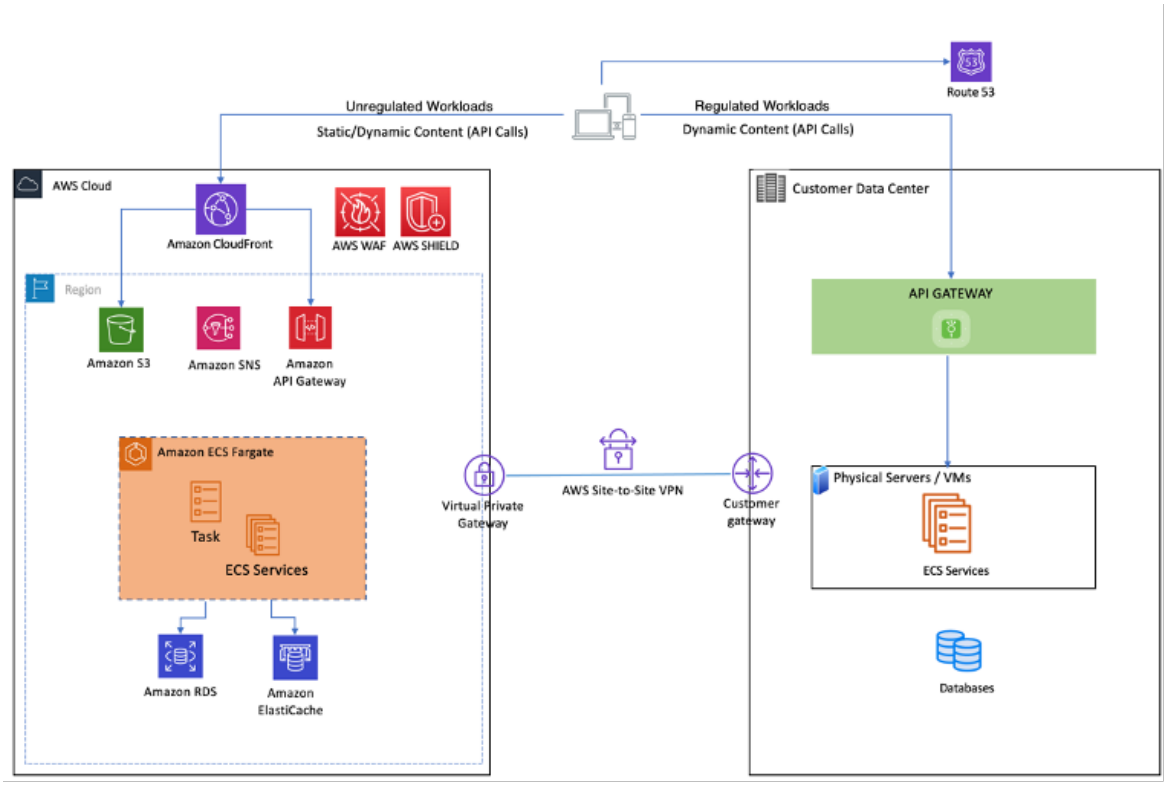
3. **Despliegue Continuo:**

- Estrategias de despliegue: Tales como canary releases, blue-green deployments o feature flags, para minimizar el riesgo de interrupciones en la producción.
- Automatización de la infraestructura: Uso de herramientas como Terraform o Ansible para automatizar la configuración de la infraestructura necesaria para los despliegues.
- Rollbacks: Implementación de mecanismos de rollback para poder revertir rápidamente un despliegue ante fallos.

4. **Colaboración y Feedback:**

- Equipos multidisciplinarios: Fomentar la colaboración entre desarrolladores, equipos de operaciones y aseguramiento de la calidad.
- Retroalimentación continua: Usar las herramientas de GitLab para recopilar feedback sobre el proceso de CI/CD y realizar mejoras continuas.

DIAGRAMA ARQUITECTÓNICO



Componentes

- API Gateway:
  - Punto único de entrada para todas las solicitudes a las APIs.
  - Gestiona el enrutamiento, la autenticación, la autorización y la limitación de velocidad.
  - Implementa un patrón de circuit breaker para aislar fallos y prevenir la propagación de errores.

- Microservicios:
  - Descomponen la aplicación en servicios más pequeños, independientes y escalables.
  - Cada microservicio se despliega en contenedores (Docker) y se gestiona con Kubernetes.
  - Se comunican entre sí a través de protocolos ligeros como HTTP o gRPC.

- Bases de Datos:
  - Se utilizan bases de datos relacionales (para datos estructurados) y NoSQL (para datos no estructurados).
  - Se replican los datos para garantizar la alta disponibilidad.

- Cloud Services:
    - Se utilizan servicios gestionados como funciones sin servidor, almacenamiento en la nube y bases de datos en la nube.
    - Se aprovecha la elasticidad y escalabilidad de la nube.

- On-Premise:
    - Se utilizan servidores físicos o virtuales para alojar componentes críticos o datos sensibles que no pueden migrar a la nube.
    - Red de Entrega de Contenidos (CDN):
    - Mejora el rendimiento y la disponibilidad de las APIs al almacenar copias de los recursos estáticos en servidores distribuidos geográficamente.

Resiliencia, Idempotencia y Escalabilidad

- Resiliencia:
    - Circuit Breaker: Evita que una API fallida afecte a otras.
    - Timeout: Establece un límite de tiempo para las solicitudes.
    - Retry: Realiza reintentos en caso de fallos temporales.
    - Bulkhead: Aisla los recursos para prevenir que un fallo afecte a todo el sistema.

- Idempotencia:
    - UUIDs: Utiliza identificadores únicos para cada solicitud.
    - Estado de la solicitud: Almacena el estado de cada solicitud en una base de datos o caché.
    - Comprobación de duplicados: Verifica si una solicitud ya ha sido procesada.

- Escalabilidad:
    - Microservicios: Descomponen la aplicación en servicios más pequeños que se pueden escalar de forma independiente.
    - Contenedores y Kubernetes: Permiten una gestión eficiente de los recursos y una alta disponibilidad.
    - Autoescalado: Ajusta automáticamente el número de instancias en función de la carga.