**Universidad de las Américas**

**Facultad de Ingeniería y Ciencias Aplicadas**

**Ingeniería en Sistemas de Computación e Informática**

**Implementación de una guía de buenas prácticas de Integración Continua en proyectos de desarrollo en el área de tecnología de una Empresa Privada.**

Edison P. Mosquera León, Carlos X. Ruiz Águila, David F. Martínez Bravo

**Pipelines**

1. **Objetivo**

Explicar el funcionamiento a nivel técnico de los pipelines, desde su arquitectura en alto nivel hasta el código de cada una de las etapas que conforman el ciclo de integración y despliegue continuo del proyecto de software.

1. **Arquitectura**

* ***Visión a alto nivel*:**

Diagrama

Descripción generada automáticamente

* Construcción**:**
* Las fuentes serán tomadas del repositorio GIT y serán construidas.
* Pruebas Unitarias:
* En caso de que el código fuente exista las pruebas unitarias serán ejecutadas en donde se mostrara el porcentaje de cobertura, en caso de que no se tengan pruebas unitarias, el pipeline mostrara una alerta.
* Calidad de Código:
* Dependiendo el caso de la tecnología usada, se compilará el código para que Sonar lo pueda revisar, dependiendo las reglas que se hayan configurado para la calidad de código, en caso de que este no cumpla las reglas, el pipeline mostrara una advertencia y sonar emitirá un informe de la calidad.
* Creación de Docker:
* El pipeline buscará el archivo de Dockerfile en donde especificaremos los diferentes pasos para que la imagen sea creada y subida a un repositorio en la nube para su posterior despliegue.
* Despliegue:
* El pipeline buscará el archivo K8s.yaml y utilizando las credenciales proporcionadas de kubernetes procederá a desplegar la imagen de Docker desde el repositorio de la nube.

1. **Descripción técnica Pipelines**

Interfaz de usuario gráfica, Word

Descripción generada automáticamente

* Trigger:

Le dice al “pipeline” si es momento de ejecutarse.

* Ejemplo:

|  |
| --- |
| trigger: [ string ] # Podemos especificar una lista de ramas. |
| trigger: none # Si el pipeline será ejecutado exclusivamente de forma manual |

* Pipeline:

Contiene uno o más “stages”, pueden ejecutarse en paralelo y pueden hacer despliegues a varios entornos.

|  |
| --- |
| variables: # Se pueden especificar variables fijas o dinámicas.  trigger: trigger # El trigger que avisa al pipeline si es momento de ejecutarse.  pr: pr # El trigger en caso de que el pipeline sea exclusivo de Pull Requests.  stages: [ stage ] # Los stages a ejecutar. |

* Stage:

Contiene uno o más “jobs”, pueden ejecutarse en paralelo y pueden hacer despliegues a varios entornos.

|  |
| --- |
| stages:  - stage: string # El nombre del stage, por ejemplo “CalidadDeCodigo”, no puede tener espacio ni símbolos especiales.  displayName: string # El nombre del stage que se mostrara al usuario, puede tener espacios.  dependsOn: string | [ string ] # Si el stage depende de mas stages, por ejemplo el stage para desplegar no puede pasar si el stage de construcción no es exitoso.  condition: string # Las condiciones en las cuales el stage va a ser ejecutado, por ejemplo podemos definir aquí solo para que ramas el stage se va a ejecutar.  variables: # Se pueden especificar variables fijas o dinámicas  jobs: [ job ] # Son los Jobs que dentro contienen los diferentes steps o tasks para ser ejecutados. |

* Job:

Cada “job” es ejecutado en un agente y contiene uno o más “steps”.

|  |
| --- |
| jobs:  - job: string # El nombre del job, por ejemplo “CalidadDeCodigo”, no puede tener espacio ni símbolos especiales.  displayName: string # El nombre del Job que se mostrara al usuario, puede tener espacios.  condition: string # Las condiciones en las cuales el stage va a ser ejecutado, por ejemplo podemos definir aquí solo para que ramas el stage se va a ejecutar.  continueOnError: boolean # Puede haber Jobs en los que se sabe que pueda fallar pero aun así queremos que el pipeline avance.  pool: pool # Especificamos la maquina virtual o también llamado agente en donde se va a ejecutar el Job.  container: containerReference # En caso de que queramos correr el Job dentro de un contenedor, aquí se especifica el nombre del contenedor a usar.  timeoutInMinutes: number # En caso de que el Job tome mucho tiempo podemos tener especificado en cuanto tiempo el Job se cancela automáticamente.    variables: # Se pueden especificar variables fijas o dinámicas.  steps: [ script | bash | powershell | task ] # Los steps que dentro contienen las diferentes lógicas para ser ejecutadas. |

* Agent:

Un “agent” o agente ejecuta un “job” .

|  |
| --- |
| pool:  name: string # Campo requerido en caso de tener pools privadas  demands: string | [ string ] # Campo requerido en caso de tener pools privadas, especifica que requerimientos debe cumplir la maquina virtual para ejecutar las tareas.  vmImage: string # Es el nombre de la imagen de la maquina virtual a usar, en este caso por default se va a usar Ubuntu-latest |

* Step:

Puede ser un “task” o un “script”, es la parte más pequeña de un pipeline.

|  |
| --- |
| steps:  - task: string # El tipo de task a correr, ejemplo: ‘Bash@3’  displayName: string # El nombre del Job que se mostrara al usuario, puede tener espacios.  condition: string # Las condiciones en las cuales el stage va a ser ejecutado, por ejemplo podemos definir aquí solo para que ramas el stage se va a ejecutar.  continueOnError: boolean # 'true' Permitir pasar al pipeline en caso de que el task falle  inputs:     script: # El script que se quiere ejecutar, EJ: echo ‘Hello World’ |

1. **Descripción técnica Pipelines .Net Core**

* analisis-codigo-net.yaml

Se encarga de compilar el microservicio y correr sonar, esto porque sonar ejecuta el análisis sobre los DLLs compilados.

|  |
| --- |
| parameters:  - name: 'sonar\_key'    type: string  stages:    - stage: Analisis\_Codigo      displayName: 'Analisis de Codigo Estatico'      pool:        vmImage: 'ubuntu-latest'      jobs:      - job: staticanalysis        displayName: 'Analisis de Codigo Estatico'  # Instala la versión de Netcore 2.0 debido a que Sonar es compatible con esta versión por ahora.        steps:        - task: UseDotNet@2          displayName: 'Instalar .Net Core'          inputs:            packageType: 'sdk'            version: '2.0.0'  # Configuramos Sonar con la conexión creada en Azure Pipelines.        - task: SonarCloudPrepare@1          displayName: 'Preparar para Escaneo'          inputs:            SonarCloud: 'SonarCloud'            organization: fullstackec            scannerMode: 'MSBuild'            projectKey: 'FullstackEC\_${{ parameters.sonar\_key }}'            projectName: ${{ parameters.sonar\_key }}  #Instalamos Netcore 3.0 pues nuestro aplicativo usa esta versión.        - task: UseDotNet@2          displayName: 'Instalar .Net Core'          inputs:            packageType: 'sdk'            version: '3.1.100'    # Restauramos las dependencias de Netcore de nuestro aplicativo.  - task: DotNetCoreCLI@2          displayName: 'Restaurar Dependencias'          inputs:            command: 'restore'  # Construimos el aplicativo.        - task: DotNetCoreCLI@2          displayName: 'Construir'          inputs:            command: 'build'  # Como se habia mencionado, para que sonar escanee el código es necesario reinstalar Netcore 2.0.        - task: UseDotNet@2          displayName: 'Instalar .Net Core'          inputs:            packageType: 'sdk'            version: '2.0.0'  # Sonar ejecuta el analisis de código basándose en los DLLs que se generaron.        - task: SonarCloudAnalyze@1          displayName: 'Correr Analisis'  # Sonar publica los resultados hacia la nube          - task: SonarCloudPublish@1          displayName: 'Publicar Analisis' |

* compilar-docker.yaml

Se genera la imagen de Docker, al nosotros usar Kubernetes es mandatorio generar imágenes de Docker para que estas luego sean desplegadas.

|  |
| --- |
| parameters:  - name: 'nombre\_microservicio'  type: string  stages:  - stage: Generar\_Docker  displayName: "Generar Imagen Docker"  dependsOn:  - Analisis\_Codigo # Dependencia del Stage de analisis de Codigo  - Verificar\_Compilacion # Dependencia del Stage de Verificar Compilacion  - Pruebas\_Unitarias # Dependencia del Stage de Pruebas Unitarias  jobs:  - job: "ContruirImagen"  steps:  # Instala Docker dentro de la maquina virtual  - task: DockerInstaller@0  inputs:  dockerVersion: "17.09.0-ce"  # Busca el dockerfile dentro del repositorio y lo construye.  - task: Docker@2  inputs:  command: "build"  Dockerfile: "\*\*/Dockerfile"  # Publica el contenedor de Docker generado hacia el registro de contenedores configurado.  - task: Docker@2  inputs:  containerRegistry: "gcp-vb-transversal"  repository: "venta-boletos-transversal/${{ parameters.nombre\_microservicio }}"  command: "buildAndPush"  Dockerfile: "\*\*/Dockerfile"  tags: |  $(Build.BuildId)  Latest |

* verificar-compilacion-net.yaml

Simplemente antes de perder tiempo con los demás stages se intenta compilar de mínima manera el código para ver si todo funciona correctamente,

|  |
| --- |
| stages:    - stage: Verificar\_Compilacion      displayName: "Verificar Compilacion"      jobs:        - job: "Restore"          steps:  # Restaura los paquetes de nuestro aplicativo con Netcore            - task: DotNetCoreCLI@2              inputs:                command: "restore"  # Contruye el aplicativo de Netcore            - task: DotNetCoreCLI@2              inputs:                command: "build" |

* despliegue-desarrollo.yaml

Al utilizar Kubernetes es muy fácil poder desplegar el aplicativo, simplemente se lee del archivo k8s.yaml las configuraciones necesarias y se le avisa a Kubernetes de que hay una nueva versión por desplegar.

|  |
| --- |
| stages:    - stage: "Despliegue\_DEV"      dependsOn: "Generar\_Docker" # Depende del stage que genera la imagen de Docker      displayName: "Despliegue Development"      condition: and(succeeded(), ne(variables['Build.SourceBranch'], 'refs/heads/master')) # Tiene la condicion de que solo si los stages anteriores son exitosos y la rama es desarrollo prosiga.      jobs:  # Se especifica un job de tipo deployment, con eso podemos registrar que se hizo un despliegue a uno de los ambientes para tener trazabilidad.        - deployment: K8s\_DEV          environment: Development          pool:            vmImage: "ubuntu-latest"          strategy:            runOnce:              deploy:                steps:                  - checkout: self                  - download: none  # Reemplaza las variables dentro del archivo de configuración de Kubernetes.                  - task: replacetokens@3                    inputs:                      targetFiles: "k8s.yaml"                      encoding: "auto"                      writeBOM: true                      actionOnMissing: "fail"                      keepToken: false                      tokenPrefix: "#{"                      tokenSuffix: "}#"  # Despliega hacia Kubernetes usando la conexión especificada con el archivo k8s.yaml                  - task: KubernetesManifest@0                    inputs:                      action: "deploy"                      kubernetesServiceConnection: "Development-default-1601140403893"                      namespace: "default"                      manifests: "k8s.yaml" |

* despliegue-test.yaml

Al utilizar Kubernetes es muy fácil poder desplegar el aplicativo, simplemente se lee del archivo k8s.yaml las configuraciones necesarias y se le avisa a Kubernetes de que hay una nueva versión por desplegar.

|  |
| --- |
| stages:    - stage: "Despliegue\_TEST"      dependsOn: "Generar\_Docker" # Depende del stage que genera la imagen de Docker      displayName: "Despliegue TEST"      condition: and(succeeded(), ne(variables['Build.SourceBranch'], 'refs/heads/master')) # Tiene la condicion de que solo si los stages anteriores son exitosos y la rama es test prosiga.      jobs:  # Se especifica un job de tipo deployment, con eso podemos registrar que se hizo un despliegue a uno de los ambientes para tener trazabilidad.        - deployment: K8s\_TEST          environment: Test          pool:            vmImage: "ubuntu-latest"          strategy:            runOnce:              deploy:                steps:                  - checkout: self                  - download: none  # Reemplaza las variables dentro del archivo de configuración de Kubernetes.                  - task: replacetokens@3                    inputs:                      targetFiles: "k8s.yaml"                      encoding: "auto"                      writeBOM: true                      actionOnMissing: "fail"                      keepToken: false                      tokenPrefix: "#{"                      tokenSuffix: "}#"  # Despliega hacia Kubernetes usando la conexión especificada con el archivo k8s.yaml                  - task: KubernetesManifest@0                    inputs:                      action: "deploy"                      kubernetesServiceConnection: "Test-default-1601140403893"                      namespace: "default"                      manifests: "k8s.yaml" |

* despliegue-prod.yaml

Al utilizar Kubernetes es muy fácil poder desplegar el aplicativo, simplemente se lee del archivo k8s.yaml las configuraciones necesarias y se le avisa a Kubernetes de que hay una nueva versión por desplegar.

|  |
| --- |
| stages:    - stage: "Despliegue\_PROD"      dependsOn: "Generar\_Docker" # Depende del stage que genera la imagen de Docker      displayName: "Despliegue PROD"      condition: and(succeeded(), ne(variables['Build.SourceBranch'], 'refs/heads/master')) # Tiene la condicion de que solo si los stages anteriores son exitosos y la rama es master prosiga.      jobs:  # Se especifica un job de tipo deployment, con eso podemos registrar que se hizo un despliegue a uno de los ambientes para tener trazabilidad.        - deployment: K8s\_PROD          environment: PROD          pool:            vmImage: "ubuntu-latest"          strategy:            runOnce:              deploy:                steps:                  - checkout: self                  - download: none  # Reemplaza las variables dentro del archivo de configuración de Kubernetes.                  - task: replacetokens@3                    inputs:                      targetFiles: "k8s.yaml"                      encoding: "auto"                      writeBOM: true                      actionOnMissing: "fail"                      keepToken: false                      tokenPrefix: "#{"                      tokenSuffix: "}#"  # Despliega hacia Kubernetes usando la conexión especificada con el archivo k8s.yaml                  - task: KubernetesManifest@0                    inputs:                      action: "deploy"                      kubernetesServiceConnection: "Production-default-1601140403893"                      namespace: "default"                      manifests: "k8s.yaml" |

* pipeline-netcore.yaml

Este pipeline simplemente se encarga de orquestar a los pipelines mencionados previamente, con eso podemos tener archivos pequeños en vez de tener un único pipeline con demasiada lógica.

|  |
| --- |
| parameters:  # Son los parametros que le pasamos al pipeline, esto hace que el pipeline sea reutilizable.  - name: 'nombre\_microservicio'    type: string  - name: 'sonar\_key'    type: string  stages:  # Aquí simplemente se cargan los templates creado previamente y se les pasa las variables que necesitan para ejecutarse.  - template: verificar-compilacion-net.yaml  - template: pruebas-unitarias-net.yaml  - template: analisis-codigo-net.yaml    parameters:      sonar\_key: $(sonar\_key)  - template: compilar-docker.yaml    parameters:      nombre\_microservicio: $(nombre\_microservicio)  - template: despliegue-desarrollo.yaml  - template: despliegue-test.yaml  - template: despliegue-produccion.yaml |