Integración Continua con GitHub Actions

Tecnologías de Servicios para Ciencia de Datos

Grado en Ciencia e Ingeniería de Datos
Universidad de Las Palmas de Gran Canaria

Introducción

En esta práctica, aprenderán a implementar un pipeline de integración continua (CI) utilizando GitHub Actions para un proyecto basado en Java. El objetivo principal es automatizar las tareas esenciales del desarrollo, como la verificación del código, la construcción del proyecto, y la validación de la calidad mediante tests unitarios.

El proyecto que se utilizará como base para esta práctica se encuentra disponible en el repositorio público de GitHub en la siguiente dirección:

`https://github.com/ulpgcjj/pathfinder`

Este repositorio contiene un servicio sencillo que utiliza grafos para calcular rutas más cortas y pesos de caminos, basado en el algoritmo de Dijkstra. Deberán clonar este repositorio como primer paso y trabajar en él para cumplir con los objetivos de la práctica.

El proyecto está basado en Maven, y el archivo `pom.xml` ya está configurado con las dependencias necesarias para compilar y ejecutar el código. Además, el proyecto incluye la integración de `JaCoCo`, una herramienta que permite medir la cobertura del código durante la ejecución de las pruebas. Esto facilita garantizar que el código desarrollado cumpla con los estándares de calidad requeridos, asegurando al menos un 80% de cobertura.

Además, se espera que configuren un pipeline que no solo automatice las tareas de construcción y prueba del código, sino que también implemente buenas prácticas de CI, como la separación de responsabilidades en diferentes `jobs` y el uso de `outputs` para la comunicación entre ellos. Al finalizar, el pipeline deberá ejecutarse correctamente al realizar un `push` en la rama `master`, garantizando al menos un 80% de cobertura en los tests unitarios.

Los objetivos de la práctica son los siguientes:

1. Clonar el repositorio del proyecto desde GitHub.
2. Crear el archivo `.github/workflows/ci.yml`.
3. Implementar los jobs especificados en este documento.
4. Configurar correctamente los pasos y la comunicación entre los jobs utilizando `outputs`.

5. Verificar que el pipeline funcione correctamente al realizar un `push` en la rama `master`.

6. Añadir los tests necesarios para cumplir con una cobertura del 80%.

Cobertura de código con JaCoCo

`JaCoCo` (Java Code Coverage) es una herramienta ampliamente utilizada en proyectos Java para medir la cobertura del código fuente. La cobertura del código es una métrica que indica qué partes del código han sido ejecutadas durante la ejecución de las pruebas. Esto permite identificar áreas del proyecto que no están siendo verificadas adecuadamente y ayuda a garantizar la calidad y confiabilidad del software. El uso de `JaCoCo` asegura que las pruebas cubran las partes críticas del código, reduciendo el riesgo de errores en producción. Esto es especialmente importante en proyectos de grafos como este, donde las operaciones complejas necesitan verificarse exhaustivamente para garantizar resultados precisos y consistentes.

En esta práctica, `JaCoCo` está configurado en el archivo `pom.xml` del proyecto. Esto significa que, al ejecutar las pruebas mediante Maven, automáticamente se generará un reporte detallado de la cobertura. Las características principales de JaCoCo son:

- **Reportes detallados.** JaCoCo genera reportes en formato HTML, XML y otros formatos, mostrando qué partes del código han sido ejecutadas y cuáles no.
- **Integración sencilla.** Funciona de manera nativa con Maven, Gradle y otras herramientas de construcción.
- **Métricas de calidad.** Permite establecer umbrales mínimos de cobertura (por ejemplo, un 80% de líneas cubiertas) y fallar el build si no se cumplen.
- **Soporte para diferentes niveles.** Ofrece cobertura a nivel de clases, métodos, líneas, ramas y bloques.

El proyecto ya tiene configurado `JaCoCo` en el archivo `pom.xml`. Esto incluye las dependencias necesarias y la configuración del plugin de Maven para generar los reportes automáticamente. Para generar un reporte de cobertura, simplemente ejecuta los siguientes comandos en el directorio raíz del proyecto:

```
mvn clean verify
```

Este comando ejecutará las pruebas configuradas en el proyecto y generará el reporte de cobertura en la siguiente ubicación:

```
target/site/jacoco/index.html
```

Abre este archivo con un navegador para visualizar un reporte detallado de la cobertura. En esta práctica, es obligatorio garantizar al menos un 80% de cobertura del código. Puedes revisar el reporte generado para identificar las áreas del código que no están siendo probadas adecuadamente y añadir las pruebas necesarias para alcanzarlo.

Además, como parte del pipeline de integración continua, se configurará un paso específico para analizar automáticamente el reporte de cobertura generado por `JaCoCo`. Este paso se encargará de leer el archivo `jacoco.xml` y verificar si la cobertura cumple

con el umbral mínimo establecido. Si no se alcanza el 80%, el pipeline fallará, lo que obligará a revisar y mejorar las pruebas antes de que el código pueda integrarse en la rama principal.

Mecanismos de intercambio entre jobs

En GitHub Actions, los jobs son independientes entre sí, lo que significa que no comparten variables, archivos o estado. Sin embargo, para construir un pipeline eficiente, necesitamos que los jobs puedan intercambiar datos y resultados de forma controlada. Esto permite que un job consuma datos generados por otro. En esta práctica, como vamos a implementar un pipeline de CI que incluye varios jobs, es importante que podamos permitir la comunicación entre ellos.

En primer lugar, para que un job pueda acceder a datos generado por otro job, debe declararlo como dependencia con la clave `needs`. Por ejemplo:

```
needs: other-job
```

Compartir archivos

Los artefactos son archivos generados por un job que pueden ser descargados o utilizados por otros jobs en el mismo workflow. Se pueden guardar y recuperar utilizando las acciones `actions/upload-artifact` y `actions/download-artifact`. Un job puede subir archivos como artefactos para que estén disponibles en otros jobs o para su descarga manual desde la interfaz de GitHub:

```
uses: actions/upload-artifact@v4
with:
  name: <artifact-name>
  path: <path/to/file>
```

Otro job puede descargar estos artefactos con:

```
uses: actions/download-artifact@v4
with:
  name: <artifact-name>
  path: <path/to/file>
```

Para ver los artefactos generados por un workflow:

1. Ve a la pestaña **Actions** en tu repositorio.
2. Selecciona el workflow correspondiente.
3. En la parte derecha de la página de la ejecución, verás una sección llamada **Artifacts** donde estarán listados los artefactos disponibles.
4. Haz clic en el nombre del artefacto para descargarlo como un archivo comprimido.

Compartir variables

Los `outputs` permiten a un job compartir pequeños valores con otros jobs que dependan de él. Un job puede definir un `output` escribiéndolo en el archivo especial `$GITHUB_OUTPUT`:

```
id: <job-id>
run: |
  echo "<key>=<value>" >> $GITHUB_OUTPUT
```

Los jobs que dependan de uno anterior pueden acceder a sus `outputs` utilizando la sintaxis:

```
${{ needs.<job-id>.outputs.<output-name> }}
```

El `job-id` corresponde al identificador único del job en el que se generan y exportan los `outputs`. Este identificador se define al declarar un job en el archivo YAML del workflow y debe ser único dentro del mismo.

Estructura del Pipeline

El pipeline de integración continua (CI) que se implementará en esta práctica sigue una estructura modular y bien definida, dividida en varios `jobs` que se encargan de tareas específicas. Cada `job` tiene una función clara dentro del flujo de trabajo, y juntos automatizan tareas esenciales del desarrollo, como la validación del proyecto, la construcción del código, y la verificación de la calidad. En este pipeline los `jobs` están interconectados mediante `needs`, lo que asegura que las tareas dependientes se ejecuten en el orden correcto. Además, los datos entre `jobs` se comparten utilizando `outputs` y artefactos.

```
name: CI Pipeline

on:
  push:
    branches: [master]
  workflow_dispatch:

jobs:
  archetype:
    runs-on: ubuntu-22.04
    steps:
      # Definir pasos aquí
    outputs:
      # Definir outputs aqui

  build:
    runs-on: ubuntu-22.04
    needs: archetype
    steps:
      # Definir pasos aquí, incluyendo subir artefactos
```

```
coverage-check:
  runs-on: ubuntu-22.04
  needs: build
  steps:
    # Descargar archivo
    # Verificar la cobertura
```

Eventos

Cada vez que se realice un cambio en la rama `master`, se ejecuta el pipeline. Adicionalmente se puede iniciar el pipeline de forma manual desde la interfaz de GitHub (evento `workflow_dispatch`).

Job: `archetype`

Este job realiza tareas preparatorias para el resto del pipeline:

- Configura el entorno de ejecución (Ubuntu 22.04 y JDK 21).
- Verifica la existencia del archivo `pom.xml`.
- Extrae las coordenadas Maven (`groupId`, `artifactId`, `version`) para que puedan ser usadas por otros jobs.

Job: `build`

Este job, que depende del anterior, se encarga de la construcción y pruebas del proyecto:

- Configura el entorno de ejecución (Ubuntu 22.04 y JDK 21).
- Restaura dependencias desde un caché para acelerar el proceso.
- Construye el proyecto utilizando Maven (`mvn clean compile test package verify`).
- Genera un archivo JAR como resultado del empaquetado.
- Genera un reporte de cobertura de código con JaCoCo.
- Guarda el archivo JAR y el reporte de cobertura como artefactos.

Job: `coverage-check`

Este job, que depende del anterior, verifica la calidad del código utilizando el reporte de cobertura:

- Configura el entorno de ejecución (Ubuntu 22.04 y JDK 21).
- Descarga el reporte de cobertura generado en el job `build`.
- Verifica que la cobertura de código cumpla un umbral mínimo (80%).

Cookbook

En esta sección, encontrarás ejemplos prácticos y configuraciones específicas para implementar los pasos necesarios en el pipeline de integración continua (CI). El objetivo del *Cookbook* es servir como una guía de referencia rápida que facilite la configuración correcta de cada paso en el archivo YAML, asegurando que los `jobs` del pipeline se ejecuten correctamente.

Esta sección proporciona ejemplos diseñados para abordar tareas comunes del pipeline, como configurar el entorno de trabajo (por ejemplo, el sistema operativo o la versión de Java), verificar la estructura del proyecto, compilar el código y ejecutar pruebas. También incluye configuraciones para utilizar `outputs` y artefactos, permitiendo la comunicación de datos entre `jobs`, y para verificar métricas de calidad, como la cobertura de código.

Cada ejemplo está diseñado para ser modular y reutilizable. Esto significa que puedes copiar los bloques de código directamente en tu archivo YAML y adaptarlos a las necesidades específicas de tu proyecto. Además, cada paso incluye una breve descripción de su propósito, junto con los comandos o configuraciones necesarias para implementarlo.

Generales

Checkout del repositorio

```
uses: actions/checkout@v4
```

Configurar JDK

```
uses: actions/setup-java@v4
with:
  java-version: '21'
  distribution: 'temurin'
```

Verificar que existe un archivo

```
run: |
  if [ -f <filename> ]; then
    echo "<filename> found"
  else
    echo "Error: <filename> not found."
    exit 1
  fi
```

Guardar archivo

```
uses: actions/upload-artifact@v4
with:
  name: <id>
  path: <path/to/file>
```

Recuperar archivo

```
uses: actions/download-artifact@v4
```

```
with:
  name: <id>
  path: <path/to/file>
```

Específicas para Java

Extraer coordenadas Maven

```
run: |
  GROUP=$(mvn help:evaluate -Dexpression=project.groupId -q -DforceStdout)
  ARTIFACT=$(mvn help:evaluate -Dexpression=project.artifactId -q -DforceStdout)
  VERSION=$(mvn help:evaluate -Dexpression=project.version -q -DforceStdout)
  echo "group=${GROUP}" >> $GITHUB_OUTPUT
  echo "artifact=${ARTIFACT}" >> $GITHUB_OUTPUT
  echo "version=${VERSION}" >> $GITHUB_OUTPUT
```

Restaurar caché de dependencias

```
uses: actions/cache@v4
with:
  path: ~/.m2/repository
  key: >
     maven-dependencies-
     {{ needs.archetype.outputs.artifact }}-
     {{ needs.archetype.outputs.version }}
```

Construir y empaquetar el proyecto

```
run: mvn clean compile test package verify
```

Verificar el umbral de cobertura

```
run: |
  THRESHOLD=80
  COVERAGE=$(grep -Po '<line-rate>\K[0-9.]*' ./reports/jacoco.xml |
    awk '{print $1 * 100}')
  if (( $(echo "$COVERAGE < $THRESHOLD" | bc -l) )); then
    echo "Coverage below threshold: $COVERAGE%"
    exit 1
  fi
  echo "Coverage meets the threshold: $COVERAGE%"
```