
Microcredencial Git
CI/CD con Github Actions

Índice

Dev vs Ops

👨‍💻 Desarrollo (Dev)

- **Objetivo:** Crear nuevas funcionalidades
- **Mentalidad:** "Cambio y velocidad"
- **Métricas:** Velocidad de desarrollo, nuevas features
- **Responsabilidad:** Hasta el deployment

🔧 Operaciones (Ops)

- **Objetivo:** Mantener sistemas estables y seguros
- **Mentalidad:** "Estabilidad y control"
- **Métricas:** Uptime, rendimiento, seguridad
- **Responsabilidad:** Producción y mantenimiento

"Funciona en mi máquina" 🤷‍♂️

"No toques nada que funcione" 🚫

Conflicto natural entre velocidad y estabilidad

DevOps

DevOps

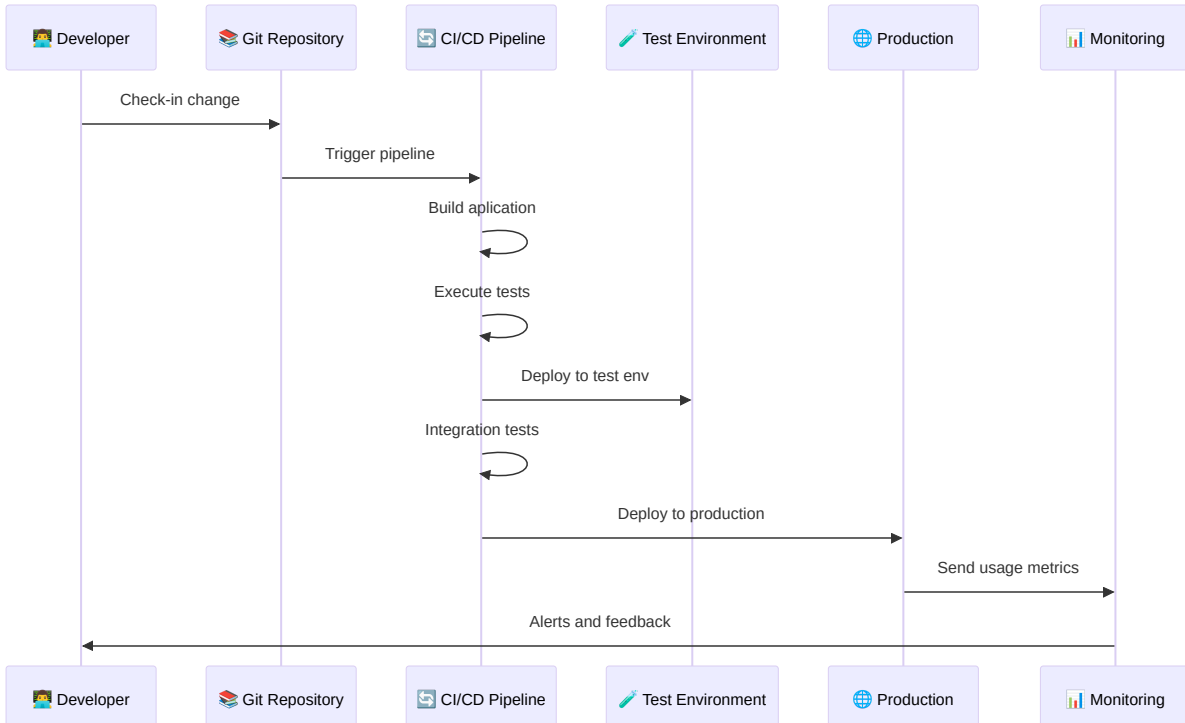
DevOps es un conjunto de prácticas, técnicas y herramientas que unifica los equipos de desarrollo de software (Dev) y operaciones (Ops).

- **Objetivo principal:**
  - Reducir tiempos de entrega de software de calidad
- **Rompe las barreras entre:**
  - Desarrollo (Dev) 👨‍💻
  - Operaciones (Ops) 🔧

Dev + Ops = DevOps

"You build it, you run it"

DevOps en la Práctica: Ejemplo



Integración Continua (CI)

+

Despliegue Continuo (CD)

🤖

Automatización total

Del commit a producción

✨ 🚀 ⚡

Herramientas del Ecosistema DevOps

🔧 Desarrollo

- Git, GitHub, GitLab
- IDE/Editors
- Jira, Trello

🏗️ Build & CI/CD

- Jenkins, GitHub Actions
- GitLab CI, Azure DevOps
- Travis CI, CircleCI

🧪 Testing

- JUnit, Jest, Selenium
- SonarQube
- Postman, Newman

📦 Containerización

- Docker
- Kubernetes
- Helm

☁️ Cloud & Infraestructura

- AWS, Azure, GCP
- Terraform, Ansible
- Vagrant

📊 Monitorización

- Prometheus, Grafana
- ELK Stack
- New Relic, Datadog

🎯 El objetivo no es usar todas las herramientas, sino elegir las que mejor se adapten a tu contexto

Integración / Despliegue Continuo

Integración Continua (CI)

Práctica de integrar cambios de código de forma frecuente en un repositorio compartido, ejecutando automáticamente la construcción y pruebas para detectar errores lo antes posible.

Despliegue Continuo (CD)

Práctica de publicar automáticamente en producción cada cambio que supera las validaciones establecidas, sin intervención manual en la fase de despliegue.

Github Actions

GitHub Actions es la plataforma de automatización de GitHub para definir workflows (CI/CD) en YAML que se ejecutan en respuesta a eventos del repositorio.

⚙️ Workflows

Un proceso que ejecuta jobs cuando ocurre un evento. Fichero YAML. Repositorio de templates.

🏃 Runners

Máquinas (en la nube o self-hosted) donde se ejecutan los jobs y sus steps. Máquinas disponibles.

📣 Events

Los workflows se activan por eventos (push, pull_request, schedule, etc.). Lista de eventos disponibles.

🧩 Jobs

Conjunto de comandos que se ejecutan. Cada job se ejecuta en una instancia runner nueva. Pueden lanzarse en paralelo (por defecto) o secuencialmente.

🪜 Steps

Pasos individuales dentro de un job. Pueden ser un shell script o un Action. Se ejecutan de manera secuencial.

🔨 Actions

Bloques reutilizables que se pueden ejecutar en un step (Marketplace).

> Cada workflow se define con un archivo `*.yml` en el directorio `.github/workflows/`. Especifica eventos, jobs, steps, etc.

Limitaciones de uso

Límites de uso según plan contratado

Precios por minuto si se superan los límites

Free use of GitHub Actions

The following amounts of time for standard runners, artifact storage, and cache storage are included in your GitHub plan. At the start of each month, the minutes used by the account are reset to zero.

Plan	Artifact storage	Minutes (per month)	Cache storage
GitHub Free	500 MB	2,000	10 GB
GitHub Pro	1 GB	3,000	10 GB
GitHub Free for organizations	500 MB	2,000	10 GB
GitHub Team	2 GB	3,000	10 GB
GitHub Enterprise Cloud	50 GB	50,000	10 GB

The use of standard GitHub-hosted runners is free:

- In public repositories
- For GitHub Pages
- For Dependabot
- For the agentic features (public preview) in GitHub Copilot code review

> **Note**
>
> - Larger runners are always charged for, even when used by public repositories or when you have quota available from your plan.
> - The storage amounts shown are **shared** with GitHub Packages. This means your total storage across Actions artifacts, Actions caches, and Packages cannot exceed the included amount for your plan.

Workflow básico

```yaml
name: Java Maven Build

on:
  push:
    branches: [ "main" ]

jobs:
  build:
    runs-on: ubuntu-latest

    steps:
      - name: Clonar repositorio
        uses: actions/checkout@v4

      - name: Configurar JDK 21
        uses: actions/setup-java@v4
        with:
          java-version: '21'
          distribution: 'temurin'
          cache: 'maven'

      - name: Compilar y empaquetar
        run: mvn clean package
```

Nombre del workflow, se verá en la pestaña Actions en GitHub.

Evento: El workflow se dispara solo al hacer push en la rama main.

Runner: Se define el job `build`, que se ejecutará en una máquina virtual Linux.

Acción `Checkout`: ejecuta `git clone`.

Instalación del JDK 21 con distribución Temurin y caché de dependencias.

Ejecución directa de Maven para compilar y empaquetar el proyecto.

Step

```yaml
- name: Empaquetar proyecto con Maven
  id: maven_package  # id opcional
  if: success() # por defecto siempre es success
  run: mvn clean package -DskipTests
  working-directory: <path>  # opcional
  shell: bash
  env:  # variables de entorno para este Step
    MAVEN_OPTS: "-Xmx1024m"
  continue-on-error: true
  timeout-minutes: 15
```

Cada `Step` se ejecuta con un proceso shell nuevo, desde el directorio raiz. Usar comando `cd` en cada Step, o establecer `working-directory` .

Pueden ejecutarse condicionalmente con `if` :

- `success()` : se ejecuta si el Step anterior fue exitoso
- `failure()` : se ejecuta si algún Step anterior falló
- `always()` : se ejecuta siempre
- `cancelled()` : se ejecuta si el workflow se canceló
- `steps.maven_package.outcome == 'failure'`

: comprobando el estado de un step concreto

Definición de variables de entorno `env` :

- A nivel del Step
- A nivel del job
- A nivel del Workflow

Ejercicio 1.1

Creación de un `Workflow` sencillo

1. Descargar el código fuente de una aplicación java sencilla en este enlace
2. Crear un repositorio en GitHub para alojar el código de esa aplicación.
3. Activar GitHub Actions en el repositorio: `Settings → Actions → General → Allow all actions`
4. Crear un `Workflow` de GitHub Actions para compilar, ejecutar las pruebas y empaquetar el programa en un archivo `jar`.
   - El Workflow tiene que ejecutarse al hacer un push en la rama `main`
   - La aplicación utiliza el gestor Maven.
     - Compilación: `mvn clean compile`
     - Ejecución de las pruebas: `mvn clean test`
     - Empaquetamiento: `mvn clean package`
5. Comprobar que el `Workflow` se ejecuta cuando se hace un push en la rama `main`

Ejercicio 1.2

Comprobar cómo se visualiza un `Workflow` que falla.

1. Crear una rama nueva, y comprobar que cuando se hace un `push` de esa rama, el `Workflow` no se ejecuta.
2. Modificar el `Workflow` para que también se ejecute en la rama nueva. Comprobar que el `Workflow` se ejecuta ahora.
3. Modificar algun fichero java para añadir algún tipo de error de compilación. Comprobar que el `Workflow` falla.
4. Rehacer el cambio anterior. Comprobar que ahora el `Workflow` funciona.
5. Modificar cualquier prueba para que no pasen correctamente.
   - Las pruebas se encuentran en el directorio `src/test/java`.
   - Buscar alguna aserción como `assertEquals(550, firstMovie.getId());`, y cambiar el valor esperado (550 por otro)
   - Comprobar que ahora el `Workflow` falla.

Ejercicio 1.3

Recuperar artefactos que se generan en GitHub Actions.

1. La ejecución de Maven genera una serie de artefactos que muestran información detallada del proceso de compilación, pruebas o empaquetado. Estos artefactos se generan en el directorio `target`
   - Comprobar que el directorio `target` esta siendo ignorado por git (fichero `.gitignore`)
2. Recuperar estos artefactos utilizando la acción `actions/upload-artifact@v4` . Tiene dos parámetros:
   - path: directorio que queremos recuperar
   - name: nombre que le vamos a dar a este artefacto (puede ser cualquiera)
3. Hacer que el `Step` que sube los artefactos sólo ocurra cuando las pruebas fallen
   - Para hacer esto, lo recomendable es tener un `Step` que únicamente ejecute las pruebas.
   - Subir sólo los artefactos localizados en el directorio `target/surefire-reports/`

Dependencias entre Workflows

Podemos lanzar un `Workflow` cuando termina otro utilizando `workflow_run`

```yaml
# Este Workflow depende del Workflow  "CI Build"
on:
  workflow_run:
    workflows: ["CI Build"]
    types: [completed]

jobs:
  run-after-build:
    if: github.event.workflow_run.conclusion == 'success'
    runs-on: ubuntu-latest
    steps:
      - name: Notificar o continuar pipeline
        run: echo "El workflow 'CI Build' finalizó correctamente."
```

- Este `Workflow` se lanzará cuando el `Workflow` `"CI Build"` termine, aunque este falle.
- Podemos poner una condicón `if` al `job` para que se ejecute sólo cuando el `Workflow` haya finalizado exitosamente.

Dependencias entre Jobs

Por defecto los `jobs` dentro de un `workflow` se ejecutan en paralelo. Con `needs` podemos establecer dependencias entre `jobs`.

```yaml
jobs:
  compile:
    runs-on: ubuntu-latest
    steps:
      - name: Compilar
        run: |
          mvn clean compile

  test:
    needs: compile
    runs-on: ubuntu-latest
    steps:
      - name: Ejecutar pruebas si la compilación funcionó
        run: |
          mvn clean test
```

- El `job` `test` se ejecuta cuando `compile` termina exitosamente.
- Podemos hacer que un `job` se ejecute cuando otro falle, añadiendo una clausula `if` al `job`
  - `- if failure()`

Matrices (`strategy.matrix`)

`strategy.matrix` permite ejecutar el mismo job varias veces, con distintas configuraciones. Típicamente para probar distintos sistemas operativos o versiones de librerías.

```yaml
jobs:
  build-and-test:
    runs-on: ${{ matrix.os }}
    strategy:
      fail-fast: false
      matrix:
        os: [ubuntu-latest, windows-latest, macos-latest]
        java: [11, 17]
    steps:
      - name: Checkout
        uses: actions/checkout@v4

      - name: Setup Java
        uses: actions/setup-java@v4
        with:
          distribution: temurin
          java-version: ${{ matrix.java }}

      - name: Build and test
        run: ...
```

- Este ejemplo ejecuta el `job` `build-and-test` 6 veces.
  - 2 versiones de Java (11, 17)
  - 3 sistemas operativos distintos (Linux, Windows, macOS)
- Por defecto, si un `job` falla, se cancela toda la matriz.
  - Para no cancelar la matriz, usar `fail-fast: false`.

Ejercicio 1.4 — Dependencias entre jobs

Crear un workflow que contenga dos jobs: `compile` y `test`.

Instrucciones:

1. En el Ejercicio 1.3, todo se ejecutaba en el mismo `job`. Ahora tenemos que modificar ese `workflow` para que haga el trabajo en varios `jobs`.
   - El `job` `compile` debe compilar el proyecto:
   - El `job` `test` debe ejecutar las pruebas. Sólo debe ejecutarse si el `job` `compile` fue exitoso. La recuperación de artefactos puede hacerse en este `job`.
2. Verificar en la pestaña Actions que `test` se ejecuta solo si `compile` finaliza correctamente.

> **NOTA** Cada job se ejecuta en una máquina virtual nueva. Este ejercicio sirve para establecer dependencias entre tareas. Pero en un caso real, separar compilación y test de esta manera no tendría sentido, porque el job de las pruebas hace la compilación.

Ejercicio 1.5 — Matrix

Crear un workflow para compilar y ejecutar pruebas en macOS, Windows y Linux usando `strategy.matrix`.

Instrucciones:

1. Modificar el `workflow` del ejercicio 1.4 para:
   - Compilar con java versión 17 y linux.
   - Ejecutar las pruebas en macOS, windows y linux, utilizando JDK versión 17, 21 y 25.
2. Comprobar cómo se muestran los `jobs` en la pestaña `Actions` del repositorio.

GitHub Releases

Publicar una versión distribuible de la aplicación dentro de GitHub

GitHub Releases

Se recomienda utilizar la acción predefinida `softprops/action-gh-release`.
Documentación

Configuración básica:

```yaml
- name: Create Release
  uses: softprops/action-gh-release@v1
  if: github.ref_type == 'tag'  # sólo ejecutarlo si se ha lanzado por la creación de un tag
  with:
    files: | # aquí van los archivos que queremos publicar en la release. Lo normal es que se creen en jobs anteriores.
      */*.zip
    body: |
      # Descripción del release. Típicamente en formato Markdown.
      # Se pueden poner múltiples líneas.
    body_path: <path> # alternativamente la descripción puede venir de un archivo
    prerelease: false
```

Ejercicio 1.6 - Creación de un Release

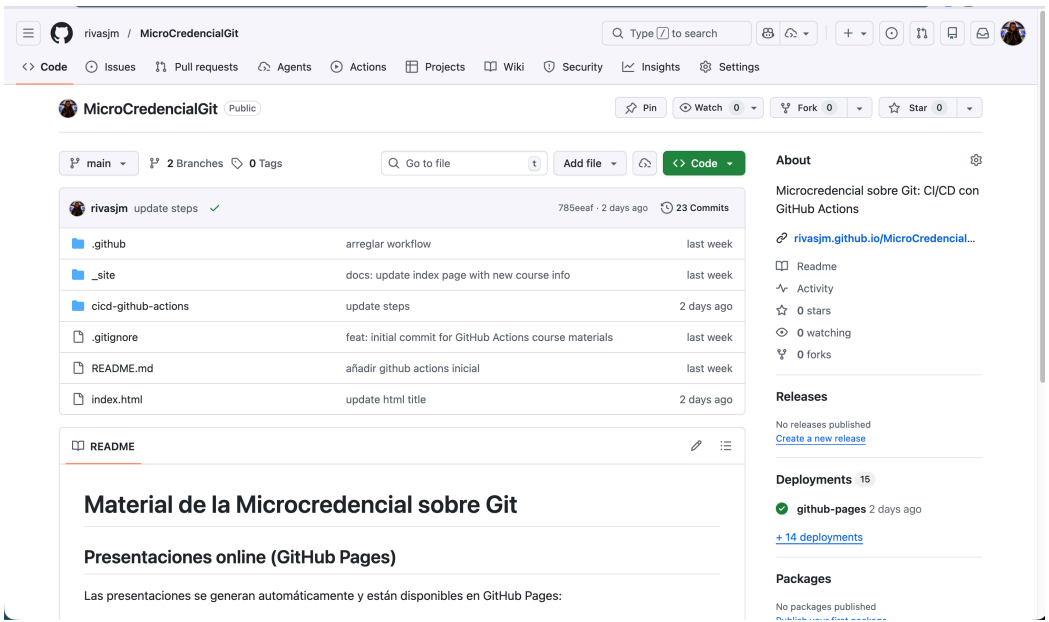
Crear un Github Release cada vez que se crea una etiqueta con formato `v*`

1. Crear un nuevo `workflow` (un nuevo fichero .yml) para hacer el *release*.
2. Para facilitar el proceso, se puede utilizar este workflow
   - El job `build-jar` construye el archivo jar (empaquetamiento de aplicaciones Java)
   - El job `build-portable` utiliza la herramienta `jpackage` para crear ejecutables de la aplicación. Utiliza `strategy.matrix` para crear ejecutables para Windows, macOS y Linux.
   - El job `create-release` utiliza el Action `softprops/action-gh-release@v1` para crear el GitHub Release.
3. Comprobar que al crear una etiqueta nueva con nombre `v*` (por ejemplo v1.0.0), se crea la Release.
4. Modificar el workflow para que el body del Release se extraiga de un fichero.
   - Hay que crear este fichero antes, y subirlo al repositorio.

Despliegue en GitHub Pages

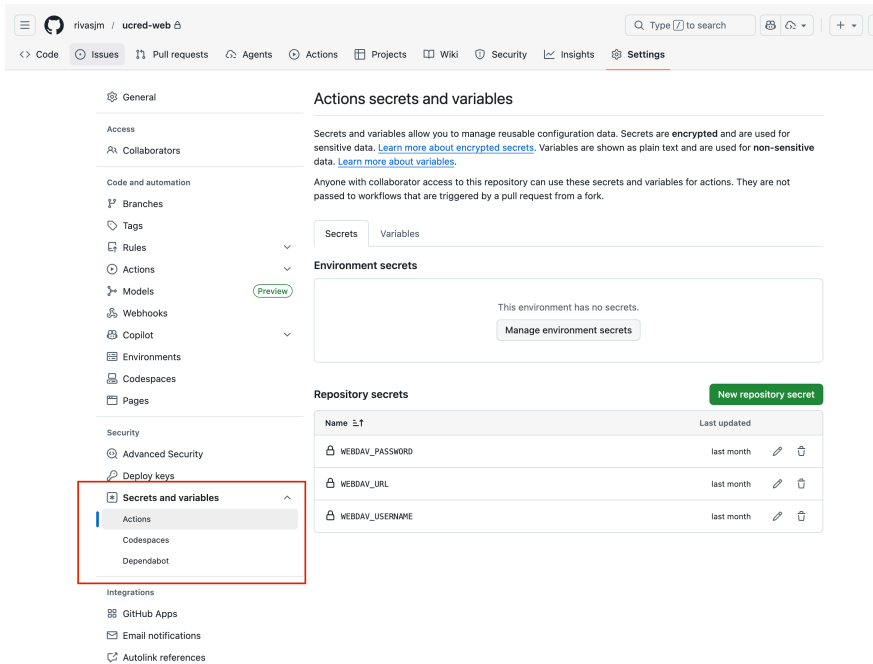
Cada repositorio en GitHub tiene disponible un espacio para alojar una página web estática mediante el servicio GitHub Pages. El despliegue a GitHub Pages se puede hacer desde GitHub Actions automáticamente.

Ejemplo: el repositorio de esta presentación: https://github.com/rivasjm/MicroCredencialGit



GitHub Secrets

Problema: ¿Y si necesitamos alguna contraseña o API Key secreta para ejecutar las pruebas o hacer el despliegue?



Esos Secrets son accesibles en el `workflow`:

- `${{ secrets.WEBDAV_PASSWORD }}`
- `${{ secrets.WEBDAV_URL }}`
- `${{ secrets.WEBDAV_USERNAME }}`

Ejercicio 1.7

Publicar una aplicación web (Calculadora) en el espacio personal de la Universidad de Cantabria

1. Descargar el código de la aplicación disponible en este link.
2. Crear un repositorio GitHub nuevo para alojar este código
3. El código ya incluye el workflow de GitHub Actions para alojar la aplicación en el espacio personal.
4. Modificar el workflow para que el despliegue se realice en vuestro espacio personal. Hay que definir los siguientes **Secrets** para que funcione:
   - `WEBDAV_URL` : url del espacio personal. Por defecto es `https://disco.unican.es/hcwebdav/Home%20Unican/www`
   - `WEBDAV_USERNAME` : tu usuario Unican completo (por ejemplo usuario@unican.es)
   - `WEBDAV_PASSWORD` : tu contraseña Unican
5. Ejecuta el workflow, y una vez que finalice comprobar que en tu espacio personal ahora hay una calculadora en el path `https://personales.unican.es/{tu-usuario}/calculadora/`
   - Ejemplo de despliegue en mi página personal

Ejecución Manual: `workflow_dispatch`

Permite lanzar workflows bajo demanda directamente desde la interfaz de GitHub sin necesidad de crear commits falsos.

```yaml
name: Deploy Manual

on:
  push:
    branches: [ "main" ]

  workflow_dispatch: # Activa el botón en la UI
    inputs:  # opcionalmente, permite definir parámetros
      entorno:
        description: 'Entorno de destino'
        required: true
        default: 'test'
        type: choice
        options:
          - test
          - prod
```

Botón "Run workflow": Al añadir `workflow_dispatch` al bloque `on`, aparece un botón en la pestaña Actions para ejecutar el workflow sin necesidad de hacer un push.

Parámetros (Inputs): Opcionalmente, puedes solicitar datos al usuario antes de la ejecución. Estos valores se utilizan en el workflow mediante la sintaxis `${{ inputs.entorno }}`.