Unidad 6

Sesión 13 : TALLER PRÁCTICO CI/CD – Pipeline de Build + Test con GitHub Actions

OBJETIVO

Vas a crear un pipeline de Integración Continua (CI) que:

- Se ejecute automáticamente al subir código a GitHub
- Instale las dependencias del proyecto
- Ejecute pruebas automáticas
- Informe si el código es correcto o contiene errores

REQUISITOS PREVIOS

Antes de empezar debes tener:

- Cuenta en GitHub
- Un repositorio vacío o nuevo

¿Qué estamos construyendo realmente?

Estamos construyendo un **sistema automático** que, cada vez que alguien sube código al repositorio:

1. **Descarga el proyecto**
2. **Prepara el entorno**
3. **Ejecuta pruebas**
4. **Dice si el código está bien (verde) o mal (rojo)**

Eso es **Integración Continua (CI)**.

Antes de mezclar cambios de varios compañeros, el repositorio se asegura de que el proyecto sigue funcionando.

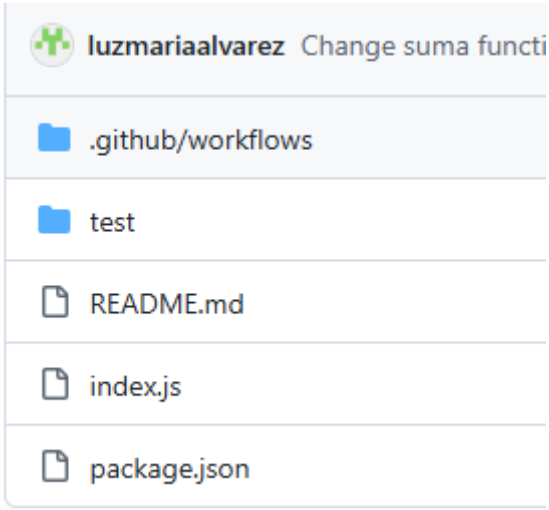
¿Por qué esto es útil?

Beneficios prácticos

- **Detecta errores en cuanto se suben**
  Si rompes algo hoy, lo sabes hoy y además te llega un aviso por email.
- **Evita 'en mi PC funciona'**
  El pipeline prueba en una máquina "limpia" y estándar (Linux).
- **Ahorra tiempo**
  No hay que repetir pruebas manuales cada vez.
- **Da seguridad al equipo**
  Si el pipeline está verde, el cambio es más confiable.
- **Crea disciplina profesional**
  Aprenden la forma real de trabajar en empresas.

PASO 1: CREAR EL REPOSITORIO

1. Accede a GitHub
2. Crea un repositorio nuevo
3. Ponle un nombre (por ejemplo: ci-practica)
4. No marques ninguna plantilla adicional



- index.js : **código que queremos validar**
- test/test.js : **pruebas automáticas**
- package.json : **instrucciones para ejecutar comandos (sobre todo tests)**
- .github/workflows/ci.yml : **pipeline: el robot que ejecuta todo en GitHub**

PASO 2: CREAR EL ARCHIVO package.json

1. En la raíz del proyecto, crea un archivo llamado:
2. package.json
3. Copia el siguiente contenido:

```
{
  "name": "proyecto-ci",
  "version": "1.0.0",
  "scripts": {
    "test": "node test/test.js"
  }
}
```

Este archivo indica cómo se deben ejecutar las pruebas del proyecto.

¿Para qué sirve?

En Node.js define:

- nombre del proyecto
- versión
- dependencias
- y lo más importante aquí: **scripts**

Qué hacemos paso a paso

Definimos un script llamado test:

"scripts": {

"test": "node test/test.js"

}

¿Por qué?

Porque el pipeline **no sabe qué archivo ejecutar**.
El pipeline ejecuta un comando estándar:

npm test

Y gracias a package.json, npm test sabe que debe ejecutar:

node test/test.js

En proyectos reales, npm test ejecuta Jest, Mocha, etc.
Aquí lo simplificamos para entender el mecanismo.

PASO 3 – CREAR EL ARCHIVO index.js

1. En la raíz del proyecto, crea el archivo:
2. index.js
3. Copia este código:

Este archivo representa el código principal del proyecto.

```
function suma(a, b) {
  return a + b;
}
module.exports = suma;
```

¿Para qué sirve?

Contiene **la lógica del programa**. En un proyecto real sería:

- tu API
- tu backend
- tu frontend
- tus funciones de negocio

Aquí usamos un ejemplo mínimo (suma) para que el foco sea CI/CD.

Qué hacemos paso a paso en este archivo

1. Escribimos la función suma(a, b)
2. Devolvemos el resultado correcto
3. Exportamos esa función para poder probarla desde el test

```
function suma(a, b) {
  return a + b;
```

```
}

module.exports = suma;
```

¿Qué demuestra este archivo?

Que **el código puede romperse**.
Si alguien cambia a + b por a - b, el proyecto compila, pero **ya no funciona**.

PASO 4 – CREAR EL ARCHIVO DE TEST

1. Crea una carpeta llamada:
2. test
3. Dentro de ella, crea el archivo:
4. test.js
5. Copia el siguiente código:

¿Para qué sirve?

Es un archivo que **comprueba automáticamente** que el código hace lo que debe.

En lugar de "probar a ojo", lo hace un test.

Qué hacemos paso a paso en este archivo

1. Importamos el código a probar:
2. const suma = require('../index');
3. Probamos un caso:
4. suma(2,2)
5. Comparamos con lo esperado:

6. if (suma(2, 2) !== 4)
7. Si está mal:
    - mostramos error
    - salimos con código 1 (fallo)
8. process.exit(1);
9. Si está bien:
    - imprimimos éxito
    - el programa termina normalmente (éxito)

Lo más importante para CI

process.exit(1) es la señal universal de:

“Esto ha fallado”

El pipeline entiende eso y marca el build como **rojo**.

```
const suma = require('../index');

if (suma(2, 2) !== 4) {
  console.error("Test fallido");
  process.exit(1);
}

console.log("Test superado");
```

Este archivo comprueba automáticamente si el código funciona correctamente.

PASO 5 – SUBIR EL PROYECTO A GITHUB

1. Guarda todos los archivos
2. Realiza un commit con un mensaje descriptivo
3. Sube los cambios al repositorio (push)

PASO 6 – CREAR LA ESTRUCTURA DEL PIPELINE

1. En la raíz del proyecto, crea la carpeta:
2. .github
3. Dentro, crea la carpeta:
4. workflows
5. Dentro de workflows, crea el archivo:
6. ci.yml

GitHub detecta automáticamente los archivos de esta carpeta como pipelines.

PASO 7 – DEFINIR EL PIPELINE CI

1. Abre el archivo ci.yml
2. Copia el siguiente contenido:

name: Pipeline CI - Build y Test

```
on:
  push:
    branches: [ main ]

jobs:
  build-test:
    runs-on: ubuntu-latest

    steps:
    - name: Clonar repositorio
      uses: actions/checkout@v3

    - name: Instalar dependencias
      run: npm install

    - name: Ejecutar tests
      run: npm test
```

Este archivo define qué se ejecuta automáticamente cuando subes código.

¿Para qué sirve?

Define el pipeline de CI dentro del repositorio.

GitHub, al ver un YAML en esa ruta, entiende:

“Esto es un workflow. Debo ejecutarlo cuando ocurra el evento indicado.”

Qué hacemos paso a paso

Nombre visible

name: Pipeline CI - Build y Test

Solo es el nombre en la interfaz de Actions.

Cuándo se ejecuta

on:

push:

branches: [main]

Significa:

- “Cada vez que alguien haga push a main, ejecútalo”

Qué trabajo se ejecuta

jobs:

build-test:

Creamos un job llamado build-test.

En qué máquina se ejecuta

runs-on: ubuntu-latest

GitHub crea una máquina virtual Linux limpia (como un servidor real).

Pasos en orden (esto es el pipeline)

steps:

Paso 1: descargar el repo

- name: Clonar repositorio

  uses: actions/checkout@v3

Sin esto, la máquina no tiene tus archivos.

Paso 2: instalar dependencias

- name: Instalar dependencias

  run: npm install

Prepara el proyecto para ejecutarse.

Paso 3: ejecutar tests

- name: Ejecutar tests

  run: npm test

Este es el paso importante:

- si los tests pasan = verde
- si fallan = rojo

PASO 8 – EJECUTAR EL PIPELINE

1. Realiza un pequeño cambio en cualquier archivo
2. Guarda
3. Haz commit
4. Sube los cambios a GitHub

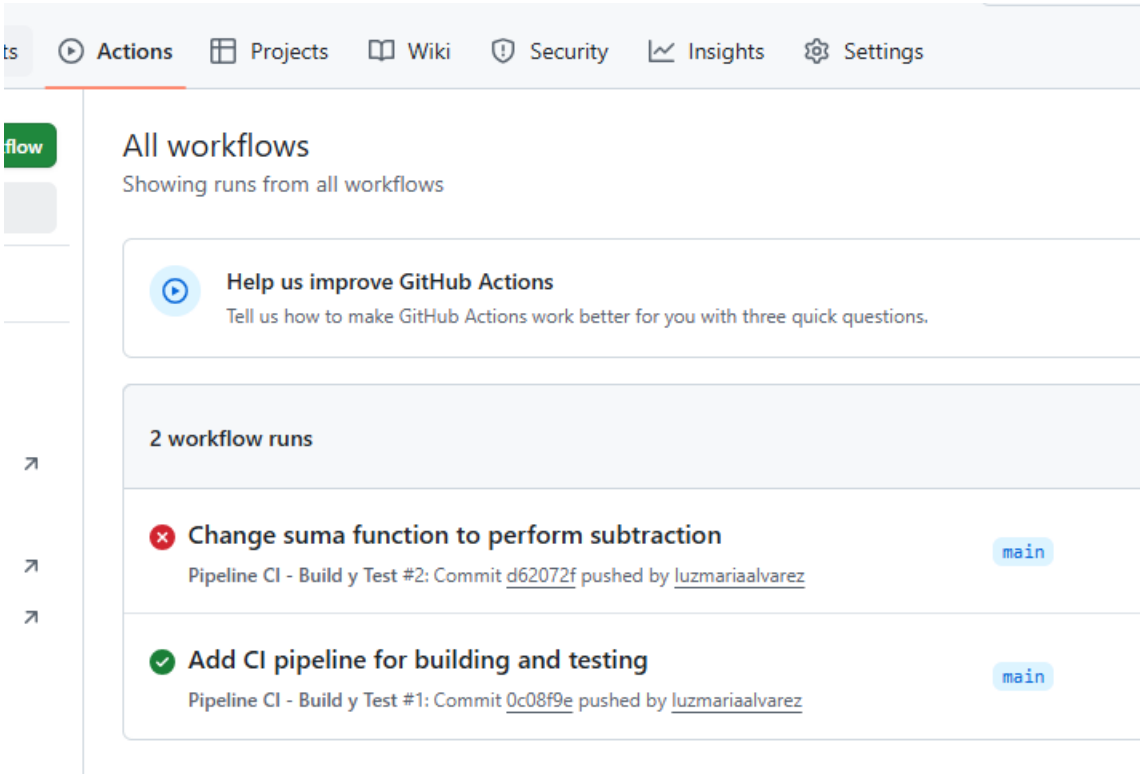
PASO 9 – COMPROBAR EL RESULTADO

1. Accede al repositorio en GitHub
2. Entra en la pestaña **Actions**

3. Observa la ejecución del pipeline

Resultados posibles:

- **Verde**: todo funciona correctamente
- **Rojo**: hay un error en el código



PASO 10 – PROVOCAR UN ERROR

1. Modifica index.js cambiando la suma:

   return a - b;

2. Guarda, haz commit y push
3. Observa cómo el pipeline falla

El sistema ha detectado automáticamente el error.

PASO 11 – CORREGIR EL ERROR

1. Vuelve a dejar el código correcto:

```
return a + b;
```

2. Haz commit y push
3. Comprueba que el pipeline vuelve a funcionar

¿Qué ocurre en cada "ejecución" del pipeline?

Cuando haces push:

1. GitHub detecta el evento (on: push)
2. Arranca una máquina Ubuntu
3. Descarga tu repo (checkout)
4. Ejecuta npm install
5. Ejecuta npm test
6. Te muestra el resultado y logs

Esto crea una trazabilidad perfecta:

- qué commit
- qué falló
- dónde falló
- por qué

¿Por qué esto es útil en un equipo real?

Ejemplo real de equipo

- Persona A cambia index.js
- Persona B cambia otra parte
- Nadie se coordina del todo
- Se sube a main

Sin CI:

- el error se descubre tarde

Con CI:

- main se mantiene “sana”
- el repositorio actúa como filtro de calidad

CI convierte el repositorio en un control de calidad automático.