DAW
Despliegue de aplicaciones web

5. Integración y despliegue continuo

Roberto Sanz Requena
rsanz@florida-uni.es

Florida
Grup Educatiu

Índice

1. Introducción

Pongamos por caso…

Un cliente pide a tu empresa de desarrollo una aplicación web con 5 funcionalidades.

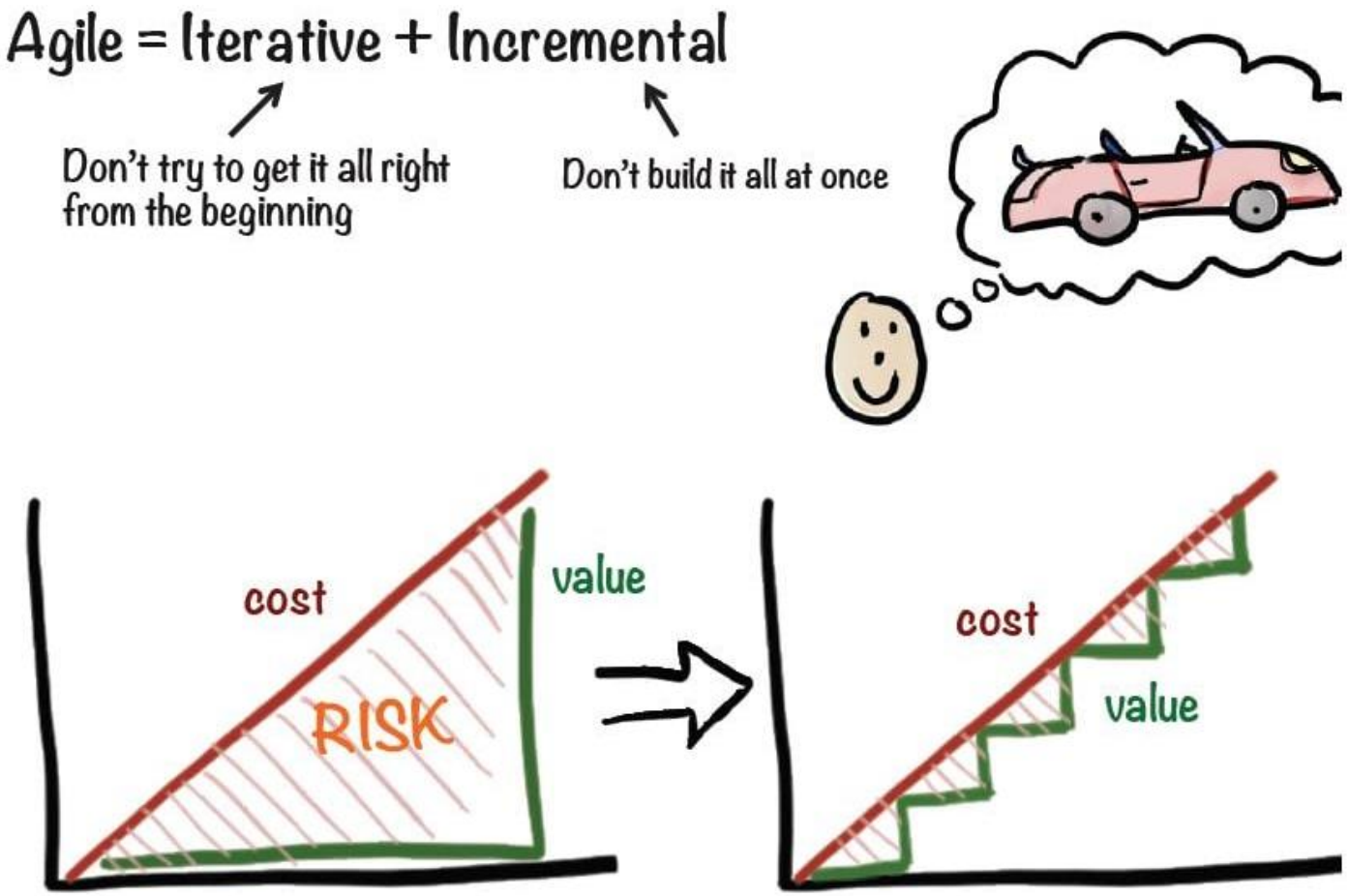
Acordáis un plazo de 2 meses para entregar una primera versión estable que pueda estar ya en producción. Es un plazo muy ajustado, pero es un cliente potencial muy importante y no quieres negociar.

Dada la importancia del cliente, destinas 4 desarrolladores (2 sénior y 2 juniors) al proyecto.

Por otro lado, en tu empresa cuentas con un titulado DAW que se suele encargar de la parte de operaciones (puesta en producción de las aplicaciones, monitorización de servidores, control de estabilidad, seguridad, etc.).

¿Cómo organizarías el trabajo?

1. Introducción



https://www.equinox.co.nz/blog/agile-practices-manage-project-risk

1. Introducción

El tipo de software también influye…

Aplicación web en producción

Al primer día de producción se detecta un error en una vista web.

Se corrige un fichero HTML en el propio servidor de la aplicación, sin necesidad de recompilar la aplicación ni reiniciar el servicio.

Coste: una disculpa y un café.

Software de la sonda Mars Climate Orbiter

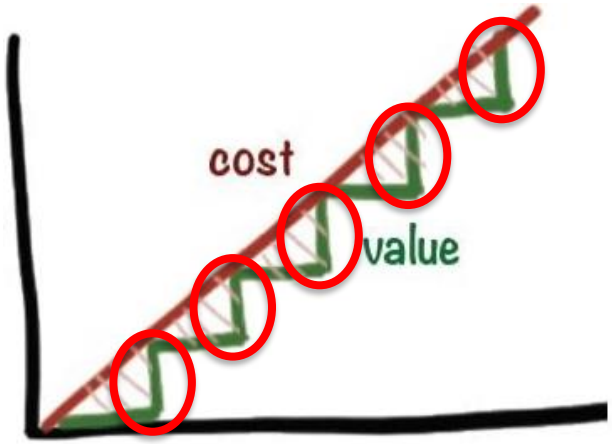
Nueve meses y medio después del lanzamiento, en vez de quedarse en la órbita de Marte, se acerca demasiado y se desintegra en la atmósfera.

El software de control terrestre funcionaba con el Sistema Anglosajón de Unidades y el de la nave con el Sistema Métrico Decimal (Sistema Internacional).

Coste: 327.6 millones de dólares (y la humillación mundial).

https://blog.cdemi.io/analyzing-software-failure-on-the-nasa-mars-climate-orbiter/

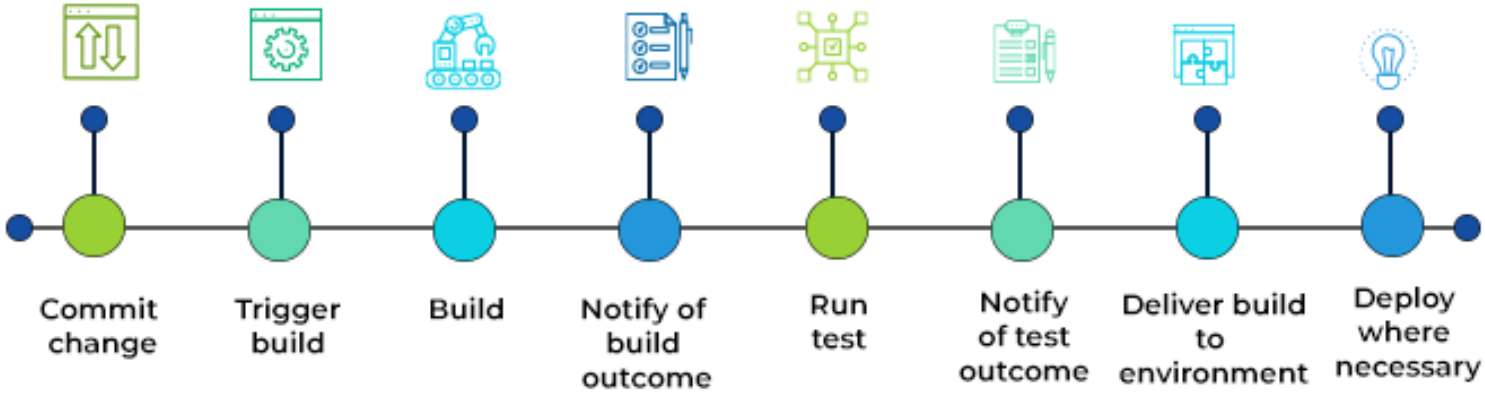
1. Introducción



Iteraciones

En cada iteración se integran los cambios que van realizando los desarrolladores y el técnico de operaciones puede ir revisando y dando *feedback* sobre los resultados del testeo y el despliegue a (pre)producción (*staging*).

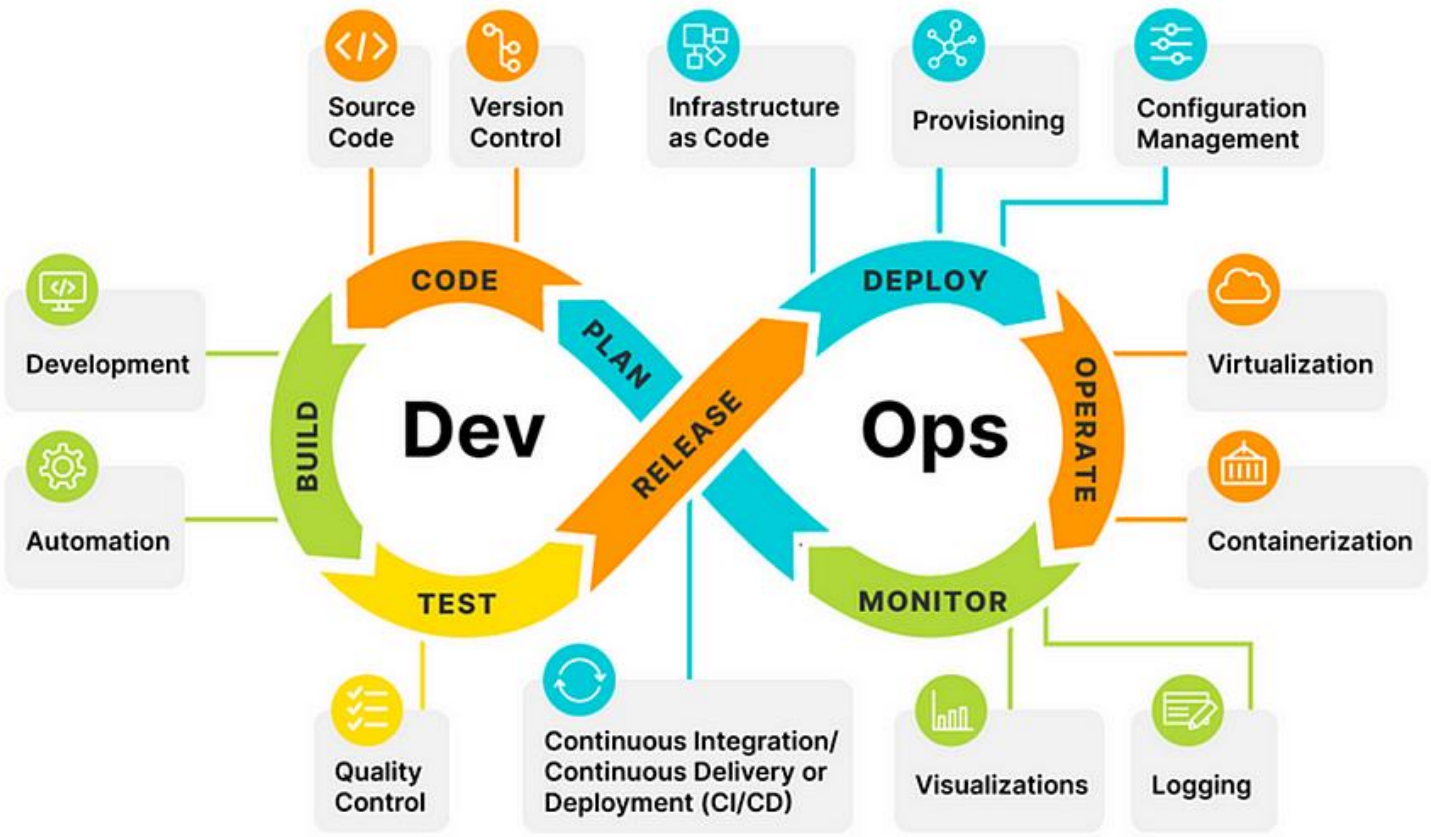
Un paso más allá ➔ Automatizar todo el proceso en una *pipeline*



https://www.spiceworks.com/tech/devops/articles/what-is-ci-cd/

1. Introducción

DevOps: combinar elementos típicos de desarrollo y de operaciones



https://zentagroup.medium.com/devops-en-la-era-de-las-ia-generativa-8a2e7badf49e

1. Introducción

Componentes clave en integración y despliegue continuo
(Continuous **I**ntegration **/ C**ontinuous **D**eployment**)**

- ✓ Repositorios de código con control de versiones: Git y GitHub/GitLab
- ✓ Pruebas automatizadas: PHPUnit (PHP), JUnit (Java), Mocha (Javascript), etc.
- ✓ Herramientas de automatización: Jenkins, Travis CI, GitHub Actions, GitLab CI/CD
- ✓ Entornos de prueba y producción: servidores en la nube (AWS, Azure, etc.).

2. Integración continua

Práctica de desarrollo de software consistente en integrar frecuentemente los cambios en el código a un repositorio común (varias veces al día).

El objetivo es detectar de forma automatizada y resolver de manera rápida los posibles errores que haya en el código.

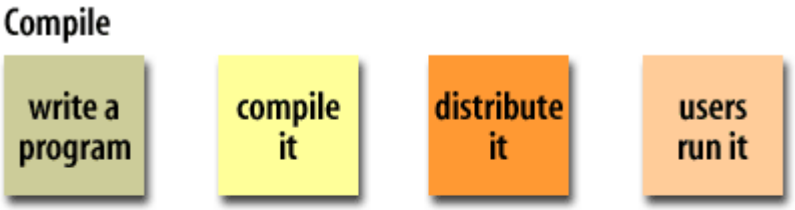
Pilares de la integración continua:

- ✓ **Control del código fuente (control de versiones):** entorno colaborativo (trabajo en ramas, *merging*, resolución de conflictos, etiquetado de versiones, etc.).
- ✓ **Compilación (*build*) automática:** compilar el código, gestionar dependencias y empaquetar el ejecutable (artefacto binario ➔ carpetas “bin”, “dist”, “target”,…).
- ✓ **Testeo automático:** pruebas unitarias, de integración, funcionales, de eficiencia, etc.
- ✓ **Seguridad**: identificar vulnerabilidades, como contraseñas débiles, configuraciones inadecuadas o dependencias obsoletas.

2. Integración continua

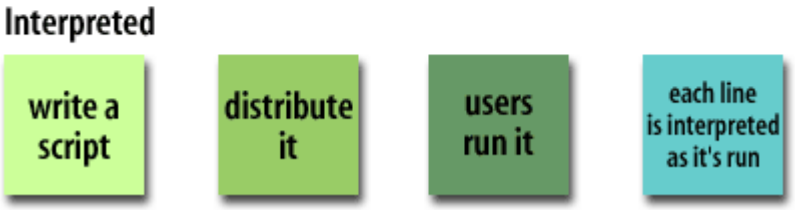
Compilación (*build*) automática

Depende del tipo de lenguaje: lenguajes compilados vs interpretados



C, C++, C#, Java,…

make dotnet maven/gradle



PHP, Javascript, Python,…

PHP es interpretado directamente por un servidor web (Apache/Nginx) a través de un módulo de PHP o a través de la línea de comandos (CLI).
Por su parte, Javascript es interpretado por el navegador.

https://airoserver.com/dedicated/interpreted-vs-compiled-languages/

2. Integración continua

Compilación (*build*) automática en lenguajes interpretados ➔ *Bundle* (paquete)

PHP
Existe la opción de empaquetar toda la aplicación en un único fichero `.phar` (el servidor Apache/Nginx debe estar configurado para ejecutar estos ficheros).
Hay que asegurarse que en el fichero quedan incluidas también las dependencias necesarias (lo puedes hacer mediante Composer).

Javascript
También permite crear un *bundle* de la aplicación mediante herramientas como `webpack`, `parcel` o `rollup`, utilizando además `npm` o `yarn` para la gestión de las dependencias del proyecto. El *framework* Angular utiliza `webpack`.

NOTA: el "bundling" no es tan común en PHP. Sin embargo, en Javascript es aconsejable, sobre todo en web frontend, ya que el código Javascript se descarga al cliente y esta estrategia supone menos solicitudes HTTP y una carga más eficiente.

https://snipcart.com/blog/javascript-module-bundler

2. Integración continua

Testeo

Utilizar herramientas y scripts para ejecutar pruebas automáticas del software.

Tipos de pruebas automatizadas:

- **Pruebas unitarias**: verifican unidades individuales de código, como funciones o métodos.
- Pruebas de integración: evalúan cómo interactúan diferentes módulos del software.
- Pruebas funcionales: aseguran que las funciones del software cumplan con los requisitos especificados.
- Pruebas de sistema: comprueban el sistema en su totalidad para verificar su comportamiento en un entorno completo.

2. Integración continua

Testeo - Pruebas unitarias con PHPUnit

PHPUnit es un *framework* de testeo para realizar pruebas unitarias en PHP.

Su funcionamiento se basa en el uso de **aserciones** (*assertions*), que son métodos para verificar que el comportamiento del código es el esperado. Una aserción compara el resultado de la ejecución del código con el resultado esperado. Si coinciden, toma el valor “true” y la prueba pasa. En caso contrario, el valor es “false” y no pasa la prueba.

Consideraciones de las pruebas unitarias:

- Solo se prueban métodos públicos de cada clase.
- No se debe hacer uso de las dependencias de la clase bajo prueba.
- El test no debe implementar lógica de negocio (if…else, for, etc.).
- La estructura de un test unitario debe ser:
  - Preparación de los datos de entrada.
  - Ejecución del test.
  - Comprobación del test (*assertion*), con solo una comprobación por test.

https://phpunit.de/

2. Integración continua

PHPUnit

Testeo - Pruebas unitarias con PHPUnit - Instalación de Composer/PHPUnit

Primero vamos a instalar **Composer** (si no lo tienes ya instalado)…
Asumiendo que tienes instalado PHP 8.2 o superior, ejecuta las siguientes instrucciones desde una ruta que reconozca el comando `php`:

```
php -r "copy('https://getcomposer.org/installer', 'composer-setup.php');"
php -r "if (hash_file('sha384', 'composer-setup.php') ===
  'dac665fdc30fdd8ec78b38b9800061b4150413ff2e3b6f88543c636f7cd84f6db9189d43a81e5503cda447da73c7
  e5b6') { echo 'Installer verified'; } else { echo 'Installer corrupt'; unlink('composer-
  setup.php'); } echo PHP_EOL;"
php composer-setup.php
php -r "unlink('composer-setup.php');"
```

La instalación habrá creado un fichero `composer.phar` en el directorio correspondiente.
Puedes comprobar que se ha instalado correctamente: `composer --version`

La idea de utilizar Composer con PHPUnit es que cada proyecto incorpore su módulo de testeo como una dependencia más en la fase de desarrollo. Por tanto, deberemos instalar PHPUnit con Composer para cada proyecto. Esto facilita el trabajo.

https://phpunit.de/
https://getcomposer.org/download/

NOTA: Si no reconoce la instrucción `composer`, puedes utilizar la instrucción `php composer.phar` en su lugar (p.ej. `php composer.phar --version`).

2. Integración continua

Testeo - Pruebas unitarias con PHPUnit - Ejemplo “Calculadora”

Crea una carpeta nueva llamada “`calculadora`”. Sitúate en la carpeta e instala PHPUnit:
`composer require --dev phpunit/phpunit`
Se han creado dos ficheros (`composer.json` y `composer.lock`) y una carpeta (`vendor`).
En la carpeta `/vendor/bin` está el ejecutable `phpunit`.

Arranca VSCode u otro editor para desarrollar el proyecto. La estructura final del proyecto deberá ser la siguiente:

```
calculadora
|-- /src
|   |-- Calculadora.php
|-- /tests
|   |-- CalculadoraTest.php
|-- /vendor
|-- composer.json
|-- composer.lock
```

https://phpunit.de/
https://getcomposer.org/download/

2. Integración continua

Testeo - Pruebas unitarias con PHPUnit - Ejemplo “Calculadora”

```
{
    "require-dev": {
        "phpunit/phpunit": "^11.4"
    },

    "autoload": {
        "psr-4": {
            "App\\": "src"
        }
    }
}
```

composer.json

```
<?php

namespace App;

class Calculadora {
    public function suma($a, $b) {
        return $a + $b;
    }
}
```

Calculadora.php

```
<?php

use PHPUnit\Framework\TestCase;
use App\Calculadora;

class CalculadoraTest extends TestCase {
    public function testSuma() {
        $calc = new Calculadora();
        $this-> assertEquals(5, $calc->suma(3,2));
    }
}
```

CalculadoraTest.php

2. Integración continua

Testeo - Pruebas unitarias con PHPUnit - Ejemplo “Calculadora”

```
{
    "require-dev": {
        "phpunit/phpunit": "^11.4"
    },

    "autoload": {
        "psr-4": {
            "App\\": "src"
        }
    }
}
```

composer.json

Indica a composer cómo cargar automáticamente las clases necesarias (estándar PSR-4)

```
<?php

namespace App;

class Calculadora {
    public function suma($a, $b) {
        return $a + $b;
    }
}
```

Espacio de nombres para agrupar clases

Calculadora.php

```
<?php

use PHPUnit\Framework\TestCase;
use App\Calculadora;

class CalculadoraTest extends TestCase {
    public function testSuma() {
        $calc = new Calculadora();
        $this->assertEquals(5, $calc->suma(3,2));
    }
}
```

Indicamos que es una clase de testeo

Importante: mismo método que en Calculadora.php pero con “test” delante (y camelCase)

Método para evaluar igualdad

CalculadoraTest.php

2. Integración continua

PHPUnit

Testeo - Pruebas unitarias con PHPUnit - Ejemplo “Calculadora”

Definido el proyecto y su código, vamos a realizar el testeo. Desde el terminal:

1. Cargar las clases necesarias con Composer: `composer dump-autoload`
2. Ejecución del test: `vendor/bin/phpunit tests`

```
PHPUnit 11.0.0 by Sebastian Bergmann and contributors.
Runtime:       PHP 8.2.12
.                                                                   1 / 1 (100%)
Time: 00:00.035, Memory: 6.00 MB
OK (1 test, 1 assertion)
```

NOTA: es conveniente acostumbrarse a ejecutar estas instrucciones desde el directorio raíz del proyecto, para evitar confusiones con las rutas a los ficheros necesarios.

2. Integración continua

Testeo - Pruebas unitarias con PHPUnit - *Asserts*

```php
<?php

use PHPUnit\Framework\TestCase;
use App\Calculadora;

class CalculadoraTest extends TestCase {
    public function testSuma() {
        $calc = new Calculadora();
        $resultado = $calc -> suma(3, 2);
        $this -> assertEquals(5, $resultado);
        $this -> assertSame(5, $resultado);
        $this -> assertGreaterThan(6, $resultado);
        $this -> assertNotNull($resultado);
    }
}
```

```
There was 1 failure:

1) CalculadoraTest::testSuma
Failed asserting that 5 is greater than 6.

...\calculadora\tests\CalculadoraTest.php:12

FAILURES!
Tests: 1, Assertions: 3, Failures: 1.
```

Para una lista completa de las pruebas que permite PHPUnit, puedes consultar la documentación (https://docs.phpunit.de/en/11.2/assertions.html).

2. Integración continua

PHPUnit

Testeo - Pruebas unitarias con PHPUnit - Desarrollo y despliegue

Es importante no perder de vista que la fase de testeo es parte del proceso de desarrollo de la aplicación, por lo que los archivos necesarios para el testeo no deben desplegarse en un entorno de producción.

Interesa, por tanto, que lo que llegue al servidor de producción sea solo lo necesario para que nuestra aplicación funcione.

Pasos:

1. Gestión del control de versiones (si desplegamos a través de GitHub)
2. Preparación local del código que irá a producción
3. Despliegue en servidor web de la aplicación “limpia”

2. Integración continua

PHPUnit

Testeo - Pruebas unitarias con PHPUnit - Desarrollo y despliegue

Gestión del control de versiones

El despliegue de la aplicación se puede configurar a partir de un *push* realizado a GitHub. Por ello, es importante que el código que llegue a GitHub esté “limpio” y preparado para producción.

Por ello, hay que utilizar el fichero `.gitignore` para indicar lo que no queremos que se suba a GitHub. Para el ejemplo de la Calculadora, deberías incluir en `.gitnore`:

```
/vendor/
/tests/
```

2. Integración continua

PHPUnit

Testeo - Pruebas unitarias con PHPUnit - Desarrollo y despliegue

Preparación local del código que irá a producción

1. Crear un directorio para el despliegue: `mkdir despliegue`

2. Copiar al directorio todos los archivos con el código fuente:
```
cp -R src despliegue
cp -R public despliegue
cp index.php despliegue
```

3. Situarse en el directorio de despliegue e instalar las dependencias de producción:
```
composer install --no-dev --optimize-autoloader
```

4. Transferir el contenido de la carpeta de despliegue al servidor correspondiente.

3. Despliegue continuo

El **despliegue continuo** es el conjunto de prácticas que permiten el proceso de entrega de cambios en el código, desde la fase de integración hasta el entorno de producción, de manera automática.

Es una forma de asegurarse que nuestra aplicación está siempre a punto de pasar a producción, de manera rápida y fiable, ya sea una versión final o una versión de prueba.

Ventajas:

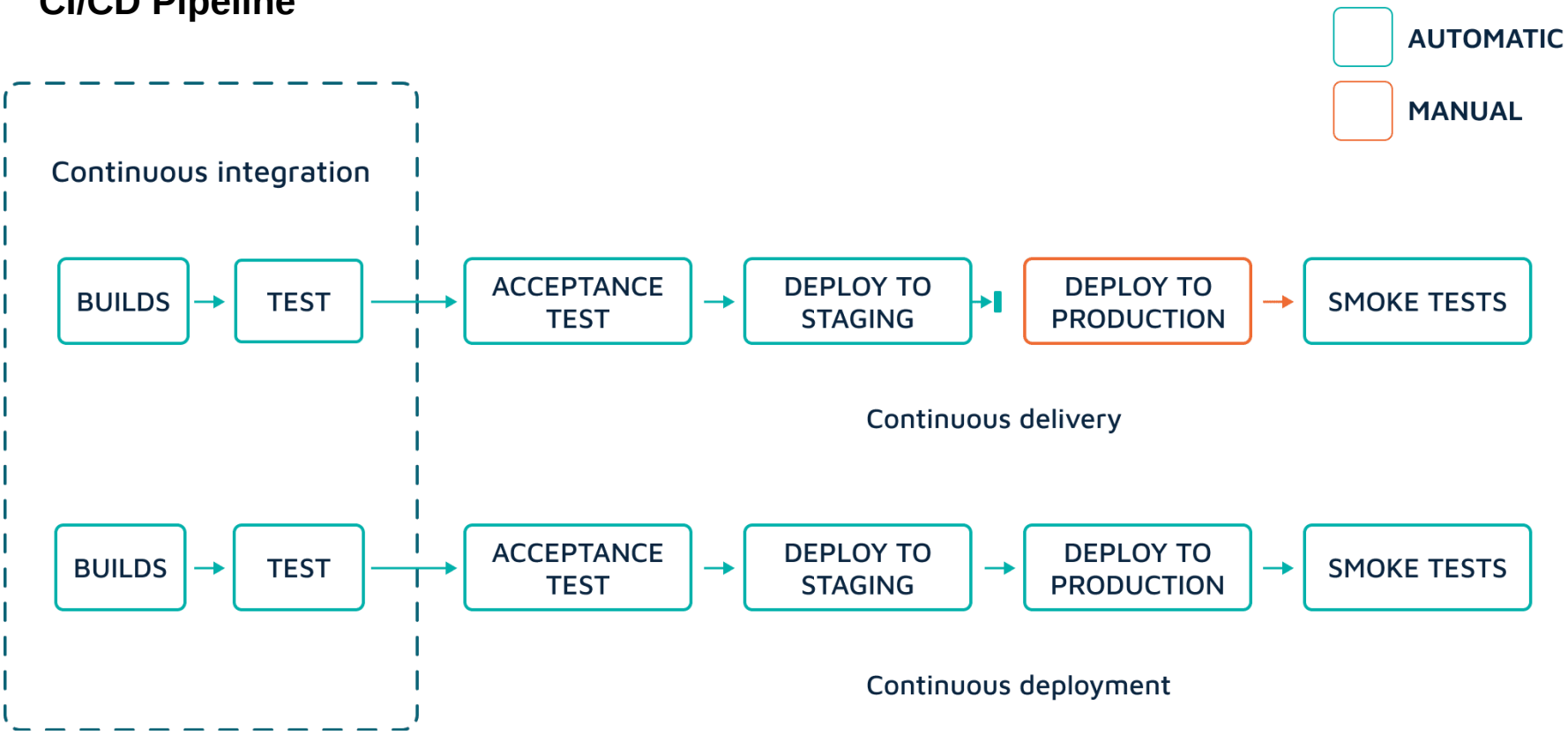
- ✓ Reducción de riesgos: cambios pequeños y frecuentes.
- ✓ Mejora la calidad del software: pruebas automatizadas.
- ✓ Acelera el paso a la fase de producción: comercialización.
- ✓ Facilita el *feedback* de los usuarios.

Entrega continua

A diferencia del despliegue continuo, en la entrega continua, el despliegue no se realiza de manera automática (por ejemplo, cada vez que se hace un *push* a GitHub), sino que se debe iniciar manualmente.

3. Despliegue continuo

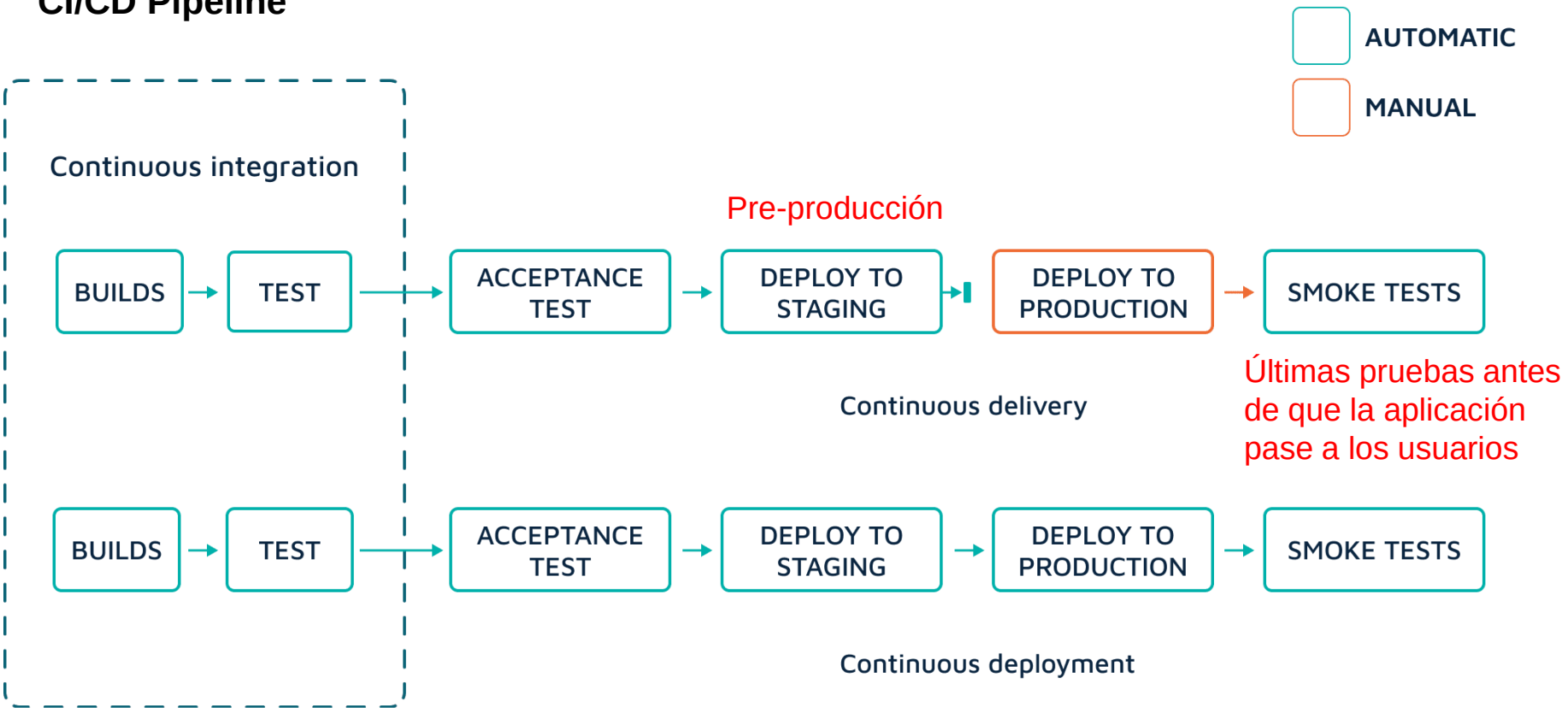
CI/CD Pipeline



https://codefresh.io/learn/continuous-delivery/continuous-deployment-only-for-unicorns/

3. Despliegue continuo

CI/CD Pipeline



https://codefresh.io/learn/continuous-delivery/continuous-deployment-only-for-unicorns/

4. Servidores de automatización

Los **servidores de automatización** son plataformas que facilitan la automatización de tareas repetitivas relacionadas con los procesos de integración continua (CI) y despliegue/entrega continuos (CD).

Se basan en scripts predefinidos que cubren la integración del código fuente, la realización de pruebas automáticas y el despliegue en diferentes entornos.

Ejemplos:

Jenkins

GitHub Actions

GitLab CI/CD

CircleCI

Travis CI

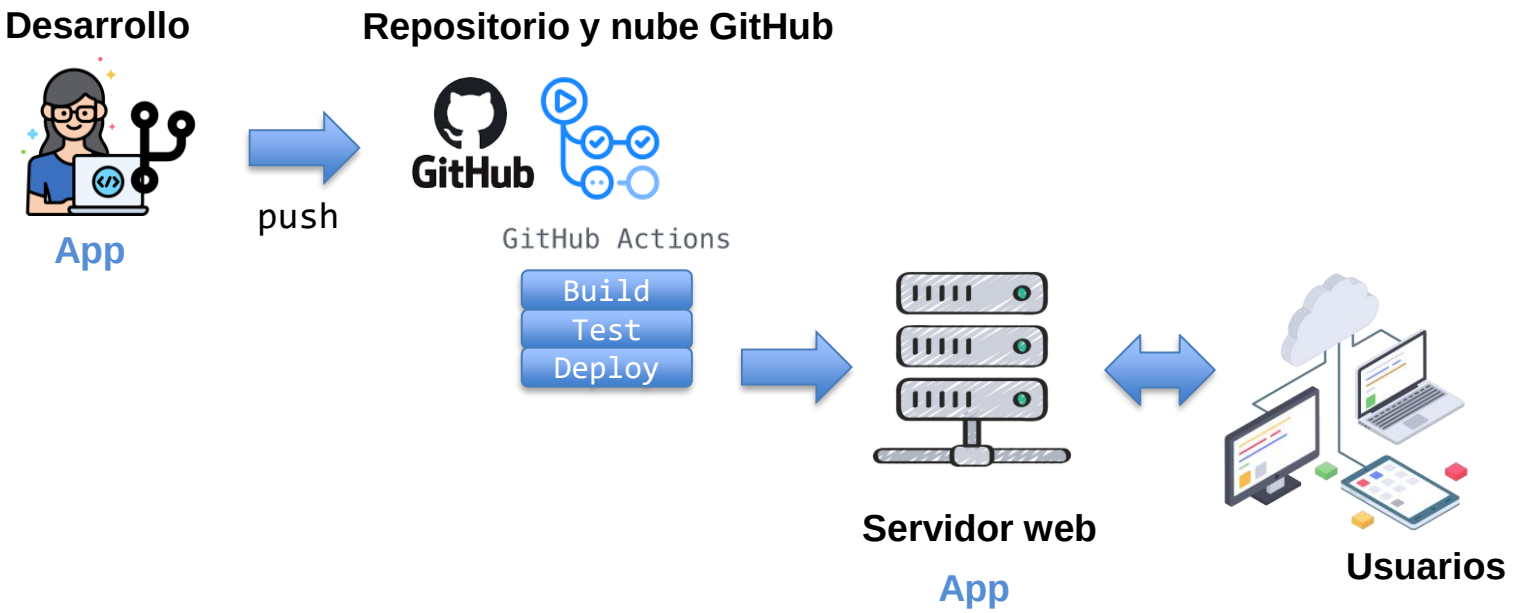
	Jenkins	GitHub Actions
Infraestructura	Gestionada por el usuario	Gestionada por GitHub
Configuración	UI y Jenkinsfile (Groovy)	Archivos YAML
Integración	Multiplataforma	Nativa con GitHub
Escalabilidad	Gestión del usuario	Gestión automática
Extensibilidad	Amplio soporte de plugins	Marketplace de acciones
Coste	Infraestructura del usuario	Gratis (repositorio público)
Curva de aprendizaje	Alta	Media

4. Servidores de automatización

GitHub Actions

GitHub Actions

GitHub Actions es una plataforma de CI/CD que permite ejecutar flujos de trabajo (compilación, testeo, despliegue, etc.) cuando sucede algún evento en el repositorio de GitHub.



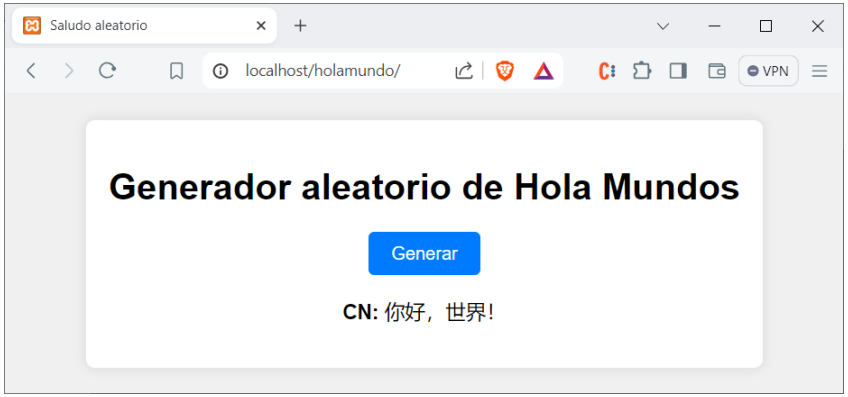
https://github.com/features/actions

4. Servidores de automatización

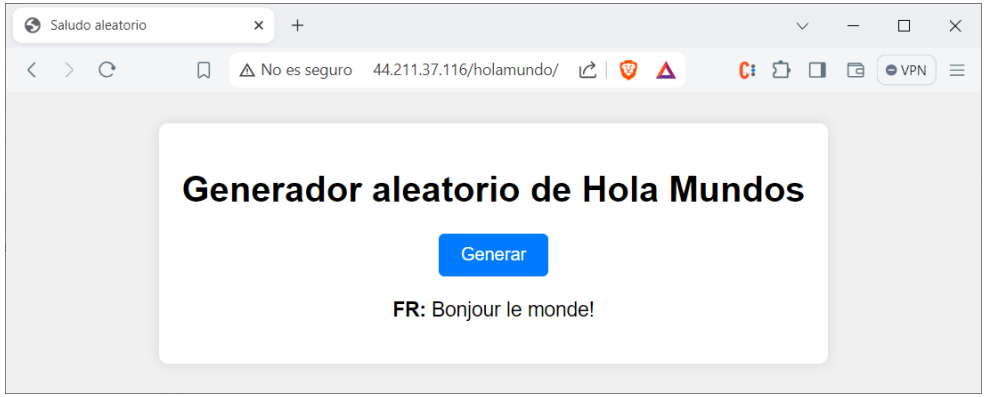
GitHub Actions

Ejemplo - Hola Mundo multilenguaje

Vamos a desarrollar una aplicación en local del tipo “Hola Mundo”, que mostrará una página con un botón. Al hacer clic, aparecerá el saludo “Hola Mundo” en un idioma aleatorio.
A través de GitHub Actions, configuraremos un *pipeline* de despliegue que se activará cada vez que se haga un *push* a la rama *main* de nuestro repositorio de GitHub. Este *pipeline* incluye una fase de testeo y una fase de despliegue, que transferirá la última versión de la aplicación (rama *main*) a un servidor web Apache alojado en una instancia EC2 de AWS con Ubuntu e IP pública.



Aplicación desarrollada en local



Aplicación desplegada automáticamente en AWS

4. Servidores de automatización

Ejemplo - Hola Mundo multilenguaje - Trabajo en local

1. Abre un terminal, crea una carpeta vacía llamada “holamundo_multi” y sitúate en ella.

2. Igual que hicimos anteriormente, instala PHPUnit: `composer require --dev phpunit/phpunit ^11` (se han creado los dos ficheros -`composer.json` y `composer.lock`- y la carpeta -`vendor`-).

3. En el directorio raíz del proyecto, crea un fichero `.gitignore` con el siguiente contenido:
   ```
   vendor/
   composer.phar
   ```

4. Crea dos carpetas en el directorio raíz: `src` y `tests`. En `src` irá el fichero de la clase que vamos a utilizar en la aplicación (`Saludo.php`) y en `tests`, la clase para testeo (`SaludoTest.php`).

5. Finalmente, crea un fichero `index.php` en el directorio raíz del proyecto.

4. Servidores de automatización

GitHub Actions

Ejemplo - Hola Mundo multilenguaje - Trabajo en local

```
{
    "require-dev": {
        "phpunit/phpunit": "11"
    },

    "autoload": {
        "psr-4": {
            "HolaMundoMultilenguaje\\": "src"
        }
    }
}
```

composer.json

4. Servidores de automatización

GitHub Actions

Ejemplo - Hola Mundo multilenguaje - Trabajo en local

```php
<?php

namespace HolaMundoMultilenguaje;

class Saludo
{
    private $saludos;

    public function __construct()
    {
        $this->saludos = [
            "es" => ["saludo" => "¡Hola Mundo!", "codigo" => "ES"], // Español
            "zh" => ["saludo" => "你好，世界！", "codigo" => "CN"], // Chino
            "en" => ["saludo" => "Hello, World!", "codigo" => "US"], // Inglés
            "hi" => ["saludo" => "नमस्ते दुनिया!", "codigo" => "IN"], // Hindi
            "ar" => ["saludo" => "مرحبا بالعالم!","codigo" => "AR"], // Árabe
            "pt" => ["saludo" => "Olá Mundo!", "codigo" => "PT"], // Portugués
            "ru" => ["saludo" => "Привет, мир!", "codigo" => "RU"], // Ruso
            "ja" => ["saludo" => "こんにちは、世界！", "codigo" => "JP"], // Japonés
            "de" => ["saludo" => "Hallo Welt!", "codigo" => "DE"], // Alemán
            "fr" => ["saludo" => "Bonjour le monde!", "codigo" => "FR"], // Francés
        ];
    }

    public function generarSaludoAleatorio()
    {
        $indice = array_rand($this->saludos);
        return $this->saludos[$indice];
    }
}
```

Saludo.php

4. Servidores de automatización

GitHub Actions

Ejemplo - Hola Mundo multilenguaje - Trabajo en local

```php
<?php

use PHPUnit\Framework\TestCase;
use HolaMundoMultilenguaje\Saludo;

class SaludoTest extends TestCase
{
    public function testGenerarSaludoAleatorio()
    {
        $saludo = new Saludo();
        $resultado = $saludo->generarSaludoAleatorio();

        // Verifica que el resultado es un string
        $this->assertIsString($resultado['saludo']);

        // Verifica que el resultado contiene un saludo válido
        $this->assertNotEmpty($resultado['saludo']);

        // Verifica que el código también es un string
        $this->assertIsString($resultado['codigo']);

        // Verifica que el código no esté vacío
        $this->assertNotEmpty($resultado['codigo']);
    }
}
```

SaludoTest.php

4. Servidores de automatización

```php
<?php

require_once __DIR__ . '/./src/Saludo.php';
use HolaMundoMultilenguaje\Saludo;
$saludo = new Saludo();
$mensaje = '';

if ($_SERVER['REQUEST_METHOD'] === 'POST') {
    $resultado = $saludo->generarSaludoAleatorio();
    $mensaje = $resultado['saludo'];
    $codigo = $resultado['codigo'];
}
?>

<!DOCTYPE html>
<html lang="es">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>Saludo aleatorio</title>
    <style>
        body { font-family: Arial, sans-serif; display: flex; justify-content: center; align-items: center;
            height: 100vh; background-color: #f0f0f0; margin: 0; }
        .container {text-align:center; background-color:#fff; padding:20px; border-radius:8px; box-shadow: 0 0 10px rgba(0,0,0,0.1);}
        button {padding: 10px 20px;font-size:16px;cursor:pointer;border:none;border-radius:5px;background-color:#007bff;color:#fff;}
        p { font-size: 18px; margin-top: 20px; }
    </style>
</head>
<body>
    <div class="container">
        <h1>Generador aleatorio de Hola Mundos</h1>
        <form method="post">
            <button type="submit">Generar</button>
        </form>
        <?php if (!empty($mensaje))
            echo "<p><b>".$codigo.":</b> ".htmlspecialchars($mensaje)."</p>";
        ?>
    </div>
</body>
</html>
```

index.php

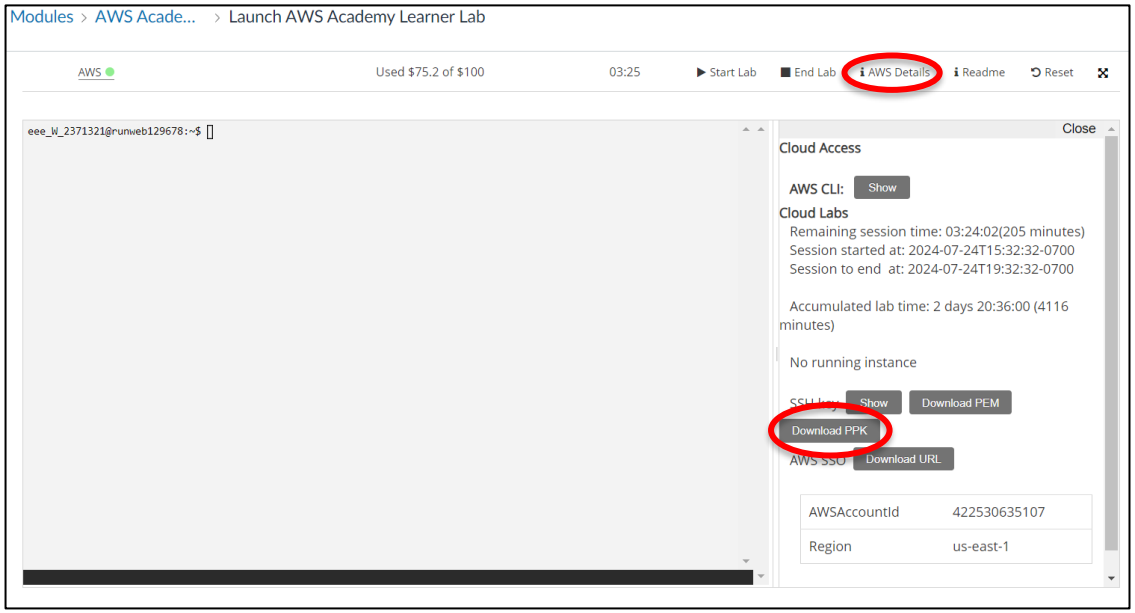
4. Servidores de automatización

GitHub Actions

Ejemplo - Hola Mundo multilenguaje - Preparación del servidor en AWS

1. Si no la has creado con anterioridad, crea una instancia EC2 (máquina virtual) de AWS con Ubuntu (puedes seguir la guía que hay disponible en Florida Oberta).

2. Desde la página de AWS Academy Learner Lab, en AWS Details, descarga la clave SSH con formato PPK (la necesitarás para acceder por PuTTy ➔ ver tema 2 ➔).



4. Servidores de automatización

Ejemplo - Hola Mundo multilenguaje - Preparación del servidor en AWS

3. Accede por SSH a la instancia EC2 (ver **tema 2**)
4. Actualiza la lista de paquetes: `sudo apt update`
5. Instala Apache: `sudo apt install apache2`
6. Instala PHP: `sudo apt install php libapache2-mod-php`
7. Habilita el módulo PHP en Apache: `sudo a2enmod php8.3` (o la versión que tengas)
8. Reinicia Apache: `sudo systemctl restart apache2`
9. Instala Composer: ver diapositivas previas
10. Prueba que el servidor funciona correctamente accediendo a la IP desde el navegador de tu PC local (debería mostrarse la página de bienvenida de Apache).

NOTA: recuerda que el directorio por defecto donde se alojará nuestra aplicación web en este servidor es `/var/www/html`.

4. Servidores de automatización

Ejemplo - Hola Mundo multilenguaje - Trabajo en GitHub

1. Crea un repositorio público en GitHub (es necesario que sea público para el plan gratuito de GitHub Actions). En este ejemplo lo he llamado `holamundomultilenguaje`.

2. En "*Settings*", accede a "*Secrets and variables* → *Actions*". Este apartado permite almacenar de manera segura variables con información sensible, como nombres de usuario, contraseñas, IPs, etc. Dentro de "*Repository secrets*", vamos a crear los siguientes:

   Name: `DEPLOY_PATH` **Secret**: ruta de despliegue (p.ej. `/var/www/html/holamundo`)
   Name: `HOST` **Secret**: IP de la instancia EC2 de AWS*
   Name: `SSH_PRIVATE_KEY` **Secret**: contenido de la clave .pem de AWS**
   Name: `USERNAME` **Secret**: nombre de usuario (p.ej. `ubuntu`)

* Ten en cuenta que si no configuras la IP pública de tu EC2 como elástica ("fija"), cada vez que inicies la EC2 cambiará, con lo que tendrás que cambiar también este secreto. Por tanto, es aconsejable configurar siempre tu máquina con IP elástica para evitar errores.

** No confundir la clave .pem con la clave .ppk, que se utiliza solo para PuTTy. El contenido del fichero .pem se debe copiar completo.

4. Servidores de automatización

Ejemplo - Hola Mundo multilenguaje - Trabajo en Local ←→ GitHub

1. En el directorio local del proyecto crea el fichero YAML (crea también la ruta) `./.github/workflows/despliegue.yml` (ver diapositiva siguiente).
2. Iniciar el repositorio: `git init`
3. Añadir el remoto: `git remote add origin https://github.com/rsanzfloridauni/holamundomultilenguaje.git`
4. Añade todos los ficheros para preparar el commit: `git add .`
5. Haz el commit: `git commit -m “Primer commit”`
6. Haz el push a GitHub: `git push -u origin main`

Si el *push* se realiza correctamente, se enviará todo al repositorio remoto y automáticamente se desencadenará la “Action” programada en el fichero `despliegue.yml`.

En GitHub, accede a la pestaña “Actions” del repositorio. Si lo haces inmediatamente después de hacer el *push*, te dará a tiempo a ver que la “Action” está aún en proceso.

despliegue.yml

```yaml
name: CI/CD Pipeline for HolaMundoMultilenguaje

on:
  push:
    branches:
      - main

jobs:
  test:
    runs-on: ubuntu-latest

    steps:
    - name: Checkout code
      uses: actions/checkout@v2

    - name: Set up PHP
      uses: shivammathur/setup-php@v2
      with:
        php-version: '8.3'

    - name: Install dependencies
      run: composer install

    - name: Run tests
      run: ./vendor/bin/phpunit tests

  deploy:
    runs-on: ubuntu-latest
    needs: test

    steps:
    - name: Checkout code
      uses: actions/checkout@v2

    - name: Set up PHP
      uses: shivammathur/setup-php@v2
      with:
        php-version: '8.3'

    - name: Install dependencies
      run: composer install

    - name: Deploy to server
      env:
        SSH_PRIVATE_KEY: ${{ secrets.SSH_PRIVATE_KEY }}
        HOST: ${{ secrets.HOST }}
        USERNAME: ${{ secrets.USERNAME }}
        DEPLOY_PATH: ${{ secrets.DEPLOY_PATH }}
      run: |
        mkdir -p ~/.ssh
        echo "$SSH_PRIVATE_KEY" > ~/.ssh/id_rsa
        chmod 600 ~/.ssh/id_rsa

        # Use SSH to create the deploy path if it doesn't exist
        ssh -o StrictHostKeyChecking=no $USERNAME@$HOST "mkdir -p $DEPLOY_PATH"

        # Sync the application files to the server
        rsync -avz --delete --no-t --exclude 'tests' --exclude '.git' . $USERNAME@$HOST:$DEPLOY_PATH

        # Optionally, run Composer install on the server
        ssh -o StrictHostKeyChecking=no $USERNAME@$HOST "cd $DEPLOY_PATH && composer install --no-dev --optimize-autoloader"

        # Restart Apache if needed
        ssh -o StrictHostKeyChecking=no $USERNAME@$HOST "sudo systemctl restart apache2"
```

```yaml
name: CI/CD Pipeline for HolaMundoMultilenguaje

on:
  push:
    branches:
      - main

jobs:
  test:
    runs-on: ubuntu-latest

    steps:
    - name: Checkout code
      uses: actions/checkout@v2

    - name: Set up PHP
      uses: shivammathur/setup-php@v2
      with:
        php-version: '8.3'

    - name: Install dependencies
      run: composer install

    - name: Run tests
      run: ./vendor/bin/phpunit tests

  deploy:
    runs-on: ubuntu-latest
    needs: test

    steps:
    - name: Checkout code
      uses: actions/checkout@v2

    - name: Set up PHP
      uses: shivammathur/setup-php@v2
      with:
        php-version: '8.3'

    - name: Install dependencies
      run: composer install

    - name: Deploy to server
      env:
        SSH_PRIVATE_KEY: ${{ secrets.SSH_PRIVATE_KEY }}
        HOST: ${{ secrets.HOST }}
        USERNAME: ${{ secrets.USERNAME }}
        DEPLOY_PATH: ${{ secrets.DEPLOY_PATH }}
      run: |
        mkdir -p ~/.ssh
        echo "$SSH_PRIVATE_KEY" > ~/.ssh/id_rsa
        chmod 600 ~/.ssh/id_rsa

        # Use SSH to create the deploy path if it doesn't exist
        ssh -o StrictHostKeyChecking=no $USERNAME@$HOST "mkdir -p $DEPLOY_PATH"

        # Sync the application files to the server
        rsync -avz --delete --no-t --exclude 'tests' --exclude '.git' . $USERNAME@$HOST:$DEPLOY_PATH

        # Optionally, run Composer install on the server
        ssh -o StrictHostKeyChecking=no $USERNAME@$HOST "cd $DEPLOY_PATH && composer install --no-dev --optimize-autoloader"

        # Restart Apache if needed
        ssh -o StrictHostKeyChecking=no $USERNAME@$HOST "sudo systemctl restart apache2"
```

Nombre del *pipeline* o *workflow*.

Evento (*trigger*) que lanza el *pipeline* o *workflow*. En este caso se lanza cuando se hace un `push` a la rama `main` (o `master`, si fuera el caso) del repositorio. Otra opción sería tener una rama específica de `deploy` que fuera solo para despliegue. O cualquier otra opción que nos venga bien para nuestro flujo de trabajo desarrollo-testeo-producción.

Jobs o tareas que se realizan en el *pipeline*: en este caso hay dos (*test* y *deploy*).

Tarea de *test*: se ejecuta en Ubuntu (última versión) y tiene varios pasos

Este paso usa la acción `checkout` para clonar el código fuente del repositorio en Ubuntu.

Este paso usa la acción `setup-php` (obtenida del repositorio de GitHub *shivamathur*) para instalar y configurar PHP en Ubuntu. Especifica que se utiliza la versión 8.3.

Este paso ejecuta el comando `composer install` en el terminal de Ubuntu para instalar las dependencias del proyecto que estén definidas en el archivo `composer.json`.

Este paso utiliza PHPUnit para ejecutar las pruebas que estén definidas en el directorio `tests`.

Tarea de *deploy*: se ejecuta en Ubuntu (última versión) y requiere para su ejecución (*needs*) que el trabajo de *test* se haya completado con éxito.

Este paso usa la acción `checkout` para clonar el código fuente del repositorio en Ubuntu.

Este paso usa la acción `setup-php` (obtenida del repositorio de GitHub *shivamathur*) para instalar y configurar PHP en Ubuntu. Especifica que se utiliza la versión 8.3.

Este paso ejecuta el comando `composer install` en el terminal de Ubuntu para instalar las dependencias del proyecto que estén definidas en el archivo `composer.json`.

Este paso ejecuta el despliegue propiamente. Primero define unas variables de entorno (*env*), que son las que previamente habrás almacenado en los secretos del repositorio.

"`run: |`" ejecuta un bloque de comandos: "`mkdir …`" crea el directorio "`~/.ssh`" si no existe; "`echo …`" guarda la clave privada SSH en un archivo "`id_rsa`"; "`chmod …`" ajusta los permisos de la clave privada para que solo el propietario pueda leer o escribir en ella.

Conexión SSH al servidor para crear el directorio de despliegue. La opción "`StrictHostKeyChecking=no`" previene interrupciones de la conexión.

```yaml
name: CI/CD Pipeline for HolaMundoMultilenguaje

on:
  push:
    branches:
      - main

jobs:
  test:
    runs-on: ubuntu-latest

    steps:
    - name: Checkout code
      uses: actions/checkout@v2

    - name: Set up PHP
      uses: shivammathur/setup-php@v2
      with:
        php-version: '8.3'

    - name: Install dependencies
      run: composer install

    - name: Run tests
      run: ./vendor/bin/phpunit tests

  deploy:
    runs-on: ubuntu-latest
    needs: test

    steps:
    - name: Checkout code
      uses: actions/checkout@v2

    - name: Set up PHP
      uses: shivammathur/setup-php@v2
      with:
```

Sincroniza (transfiere) archivos entre repositorio y servidor con las siguientes opciones:
`-a`: preserva propiedades de los archivos;
`-v`: muestra información de progreso;
`-z`: comprime datos para la transferencia;
`--delete`: borra archivos en destino que no existan en el repositorio;
`--no-t`: no preserva tiempos de modificación (puede dar problemas en el servidor)
`--exclude 'tests' –exclude '.git'`: excluyen los directorios `tests` y `.git` de la sincronización (no estarán en el código desplegado)

```
        echo "$SSH_PRIVATE_KEY" > ~/.ssh/id_rsa
        chmod 600 ~/.ssh/id_rsa

        # Use SSH to create the deploy path if it doesn't exist
        ssh -o StrictHostKeyChecking=no $USERNAME@$HOST "mkdir -p $DEPLOY_PATH"

        # Sync the application files to the server
        rsync -avz --delete --no-t --exclude 'tests' --exclude '.git'    $USERNAME@$HOST:$DEPLO

        # Optionally, run Composer install on the server
        ssh -o StrictHostKeyChecking=no $USERNAME@$HOST "cd $DEPLOY_PATH && composer install --no-dev --optimize-autoloader"

        # Restart Apache if needed
        ssh -o StrictHostKeyChecking=no $USERNAME@$HOST "sudo systemctl restart apache2"
```

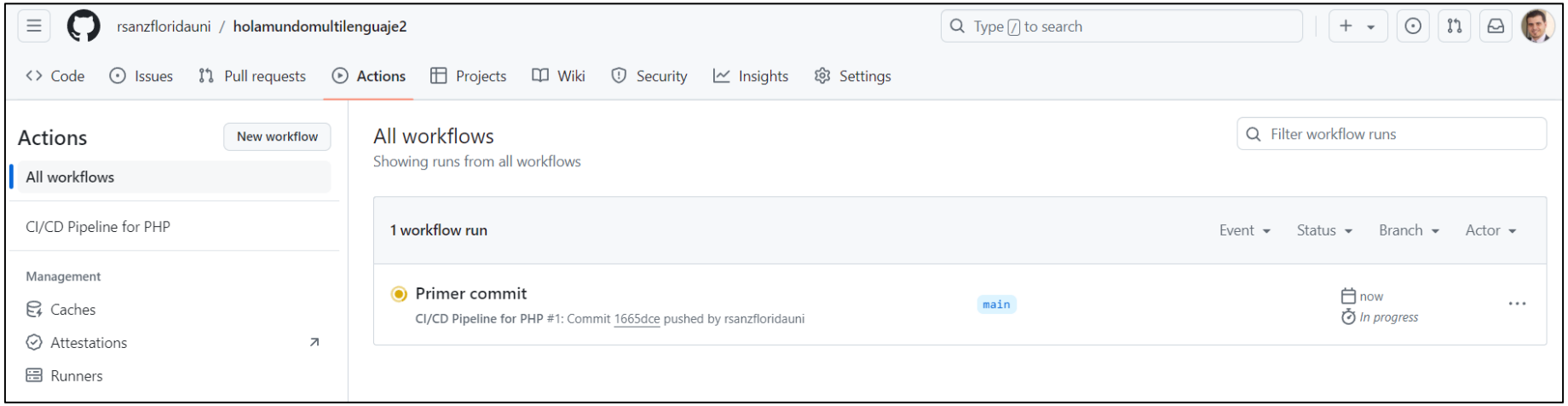
Conexión vía SSH al servidor y ejecución de la instrucción `composer install` en el directorio de despliegue, pero excluyendo las dependencias de desarrollo (`--no-dev`) (no instala PHPUnit, en este caso) y optimizando el cargador automático.

Conexión vía SSH al servidor y reinicio del servidor Apache para aplicar cualquier cambio en la configuración o en los archivos.

4. Servidores de automatización

GitHub Actions

Ejemplo - Hola Mundo multilenguaje - GitHub Action en proceso

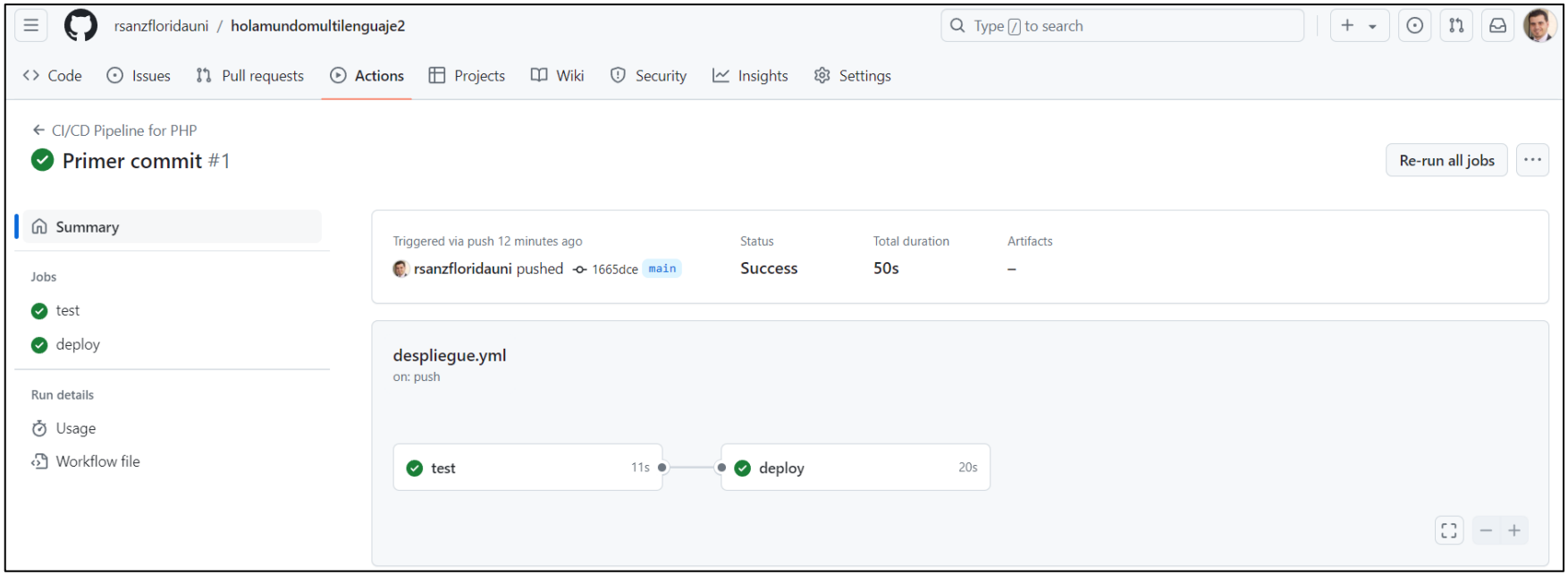


Esta captura muestra el *workflow* justo después de hacer el *push* desde local a GitHub. Se puede ver que está en proceso (“*In progress*”) y que aparece identificado por el *commit* que hemos hecho (“Primer commit”).

4. Servidores de automatización

GitHub Actions

Ejemplo - Hola Mundo multilenguaje - GitHub Action finalizada



Worfklow finalizado. Al hacer clic en el título del *commit* (“Primer *commit*”) se muestra el detalle. Se puede ver que aparecen dos partes: *test* y *deploy*, que se corresponden con los *jobs* que se indicaban en el fichero `despliegue.yml`. En este caso, ambos han finalizado correctamente, por lo que aparecen en verde. En caso de error, aparecerían en rojo y al hacer clic en ellos nos mostrarían el detalle del error.

4. Servidores de automatización

GitHub Actions

Ejemplo - Hola Mundo multilenguaje - GitHub Action finalizada

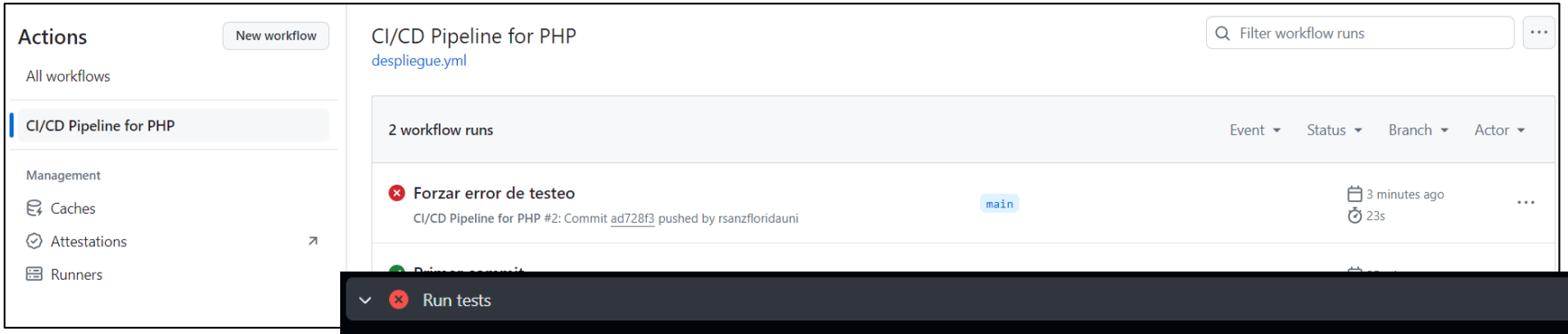
Detalle del *job* de *test*, donde se muestran los tests realizados de manera correcta.
Para el *job* de *deploy* también se puede ver todo lo que se ha realizado automáticamente.

4. Servidores de automatización

GitHub Actions

Ejemplo - Hola Mundo multilenguaje - Error de testeo

Puedes probar de forzar un error de testeo para ver que el *workflow* da error y no termina. En este caso, he modificado la última línea de `SaludoTest.php` para que compruebe que `$resultado['código']` sea vacío (`assertEmpty`). Hago `commit` y `push`, y ahora vemos que el proceso no acaba correctamente y, por tanto, no hay despliegue.



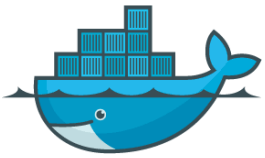
4. Servidores de automatización

Ejemplo - Hola Mundo multilenguaje

Al finalizar, el proyecto deberá tener la siguiente estructura:

```
/holamundo
|--  /.github
|    |--  /workflows
|         |-- despliegue.yml
|--  /src
|    |-- Saludo.php
|--  /tests
|    |-- SaludoTest.php
|--  /vendor
|--  .gitignore
|--  composer.json
|--  composer.lock
|--  index.php
```

4. Servidores de automatización

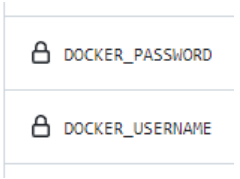


Ejemplo - Hola Mundo multilenguaje - Automatizar imágenes Docker

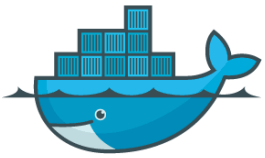
Como un plus al despliegue realizado, también puedes automatizar que se cree una imagen Docker de tu aplicación y se guarde en Docker Hub.

Hay que realizar los siguientes pasos:

1. En el repositorio, crear un fichero `Dockerfile` con las instrucciones para crear una imagen personalizada a partir del código de la aplicación.
2. En GitHub, añadir los secretos necesarios para iniciar sesión en Docker Hub.
3. En el fichero YAML, crear la imagen a partir del fichero `Dockerfile`.
4. En el fichero YAML, iniciar sesión en Docker Hub.
5. En el fichero YAML, enviar la imagen a Docker Hub.



4. Servidores de automatización

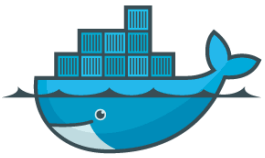


Ejemplo - Hola Mundo multilenguaje - Automatizar imágenes Docker

Dockerfile

```
FROM php:8.3-apache

# Instala extensiones de PHP necesarias
RUN docker-php-ext-install mysqli pdo pdo_mysql

# Copia el código fuente de tu aplicación al contenedor
COPY . /var/www/html/

# Establece los permisos adecuados para el directorio de la aplicación
RUN chown -R www-data:www-data /var/www/html
RUN chmod -R 755 /var/www/html

# Expone el puerto 80 para que Apache pueda recibir tráfico web
EXPOSE 80

# Comando para mantener el contenedor en ejecución mientras Apache funcione
CMD ["apache2-foreground"]
```

4. Servidores de automatización

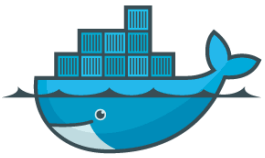


Ejemplo - Hola Mundo multilenguaje - Automatizar imágenes Docker

Añadir al final del fichero `despliegue.yml` (en el bloque `deploy`)

```
- name: Build Docker image
  run: docker build -t ${{ secrets.DOCKER_USERNAME }}/hola-mundo-multilenguaje:${{ github.sha }} .

- name: Log in to Docker Hub
  uses: docker/login-action@v2
  with:
    username: ${{ secrets.DOCKER_USERNAME }}
    password: ${{ secrets.DOCKER_PASSWORD }}

- name: Push Docker image to Docker Hub
  run: docker push ${{ secrets.DOCKER_USERNAME }}/hola-mundo-multilenguaje:${{ github.sha }}
```

4. Servidores de automatización



Ejemplo - Hola Mundo multilenguaje - Automatizar imágenes Docker

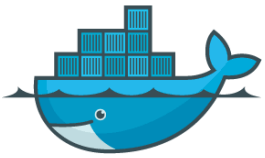
Añadir al final del fichero `despliegue.yml` (en el bloque `deploy`)

```
- name: Build Docker image
  run: docker build -t ${{ secrets.DOCKER_USERNAME }}/hola-mundo-multilenguaje:${{ github.sha }} .

- name: Log in to Docker Hub
  uses: docker/login-action@v2
  with:
    username: ${{ secrets.DOCKER_USERNAME }}
    password: ${{ secrets.DOCKER_PASSWORD }}

- name: Push Docker image to Docker Hub
  run: docker push ${{ secrets.DOCKER_USERNAME }}/hola-mundo-multilenguaje:${{ github.sha }}
```

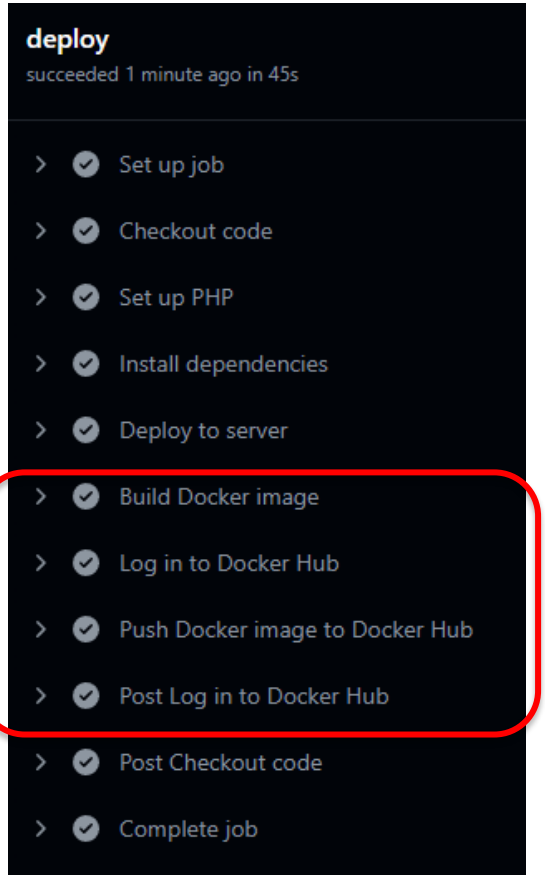
Nombre de la imagen

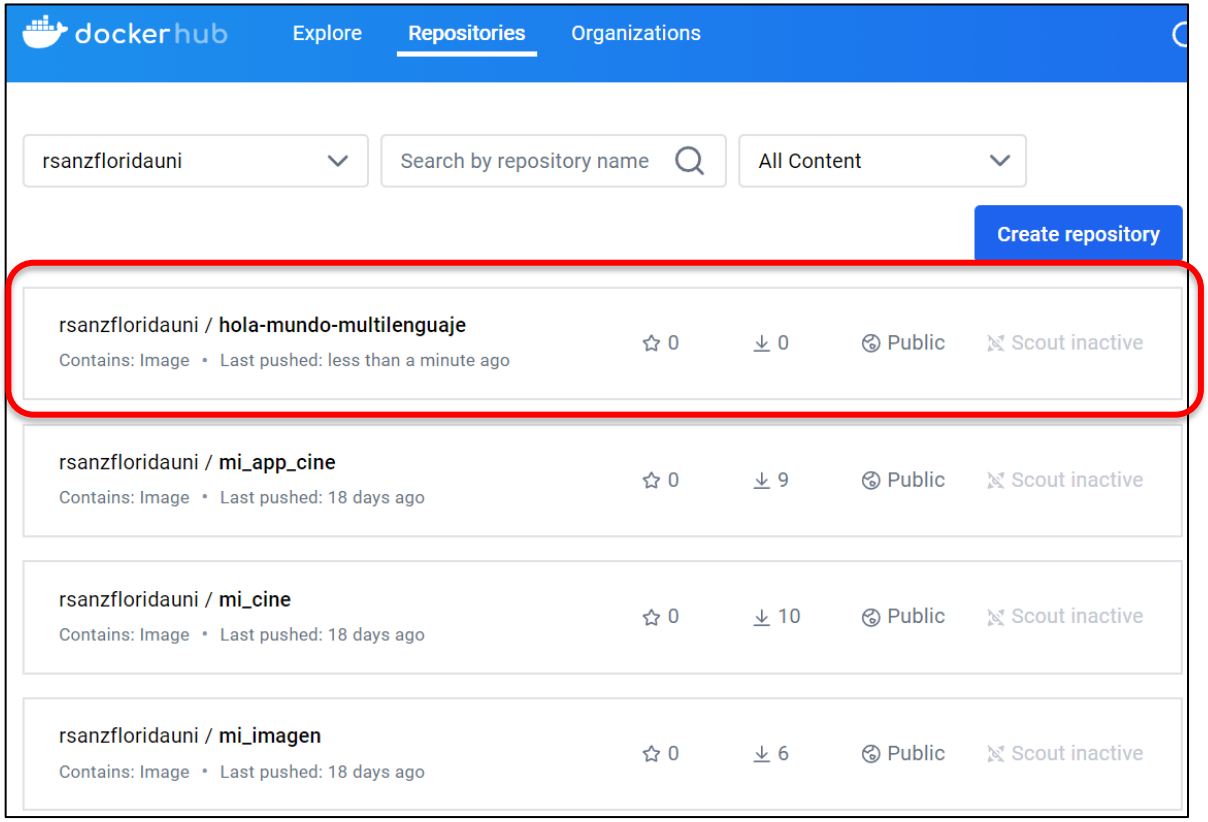
NOTA: `github.sha` es el identificador del *commit* que activa el pipeline de GitHub Actions. Es un identificador único de GitHub que resulta práctico para etiquetar la versión de imagen de Docker que se generó a partir de este *commit*. De esta forma se facilita la identificación y la trazabilidad.

4. Servidores de automatización

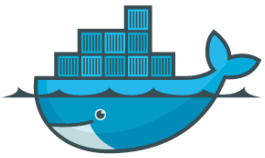


Ejemplo - Hola Mundo multilenguaje - Automatizar imágenes Docker





4. Servidores de automatización



Automatizar despliegue con Docker

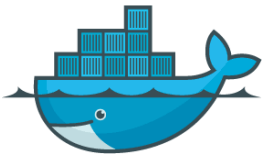
Vamos a construir un ***workflow* (o *pipeline*) nuevo** basado solamente en Docker, es decir, no hay que instalar Apache, PHP, MySQL,… en el servidor, solo hay que instalar Docker y Docker Compose y asegurarse de que los puertos SSH (22) y de la aplicación (p.ej. 8080, 8081,…) estén abiertos.

Pasos:

- En la máquina remota: instalar Docker y Docker Compose y abrir puerto SSH.
- En el repositorio de GitHub: crear los secretos correspondientes.
- En local:
  - Desarrollar la aplicación: en este caso basada en PHP y MySQL.
  - Crear un Dockerfile personalizado para generar la imagen de la aplicación.
  - Crear un Dockerfile personalizado para generar la imagen de la base de datos.
  - Crear un fichero Docker Compose para orquestar los servicios.
  - Crear el fichero .github/workflows/deploy.yml.

NOTA: Por simplicidad vamos a prescindir en este ejemplo de la parte de testeo automático.

4. Servidores de automatización



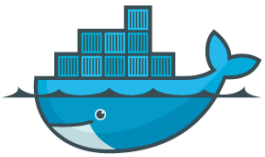
Automatizar despliegue con Docker - Trabajo en la máquina remota

Pasos:

- En la máquina remota: instalar Docker y Docker Compose (igual que en el tema de Contenedores) y abrir puertos SSH, 8080 y 8081 (revisar la guía de AWS disponible en el F.O.).

```
sudo apt-get update
sudo apt-get upgrade
sudo apt-get install apt-transport-https ca-certificates curl software-properties-common
curl –fsSL https://download.docker.com/linux/ubuntu/gpg | sudoapt-key add -
sudo add-apt-repository "deb [arch=amd64] https://download.docker.com/linux/ubuntu $(lsb_release-cs) stable"
sudo apt-get update
sudo apt-get install docker-ce
sudo systemctl status docker
```

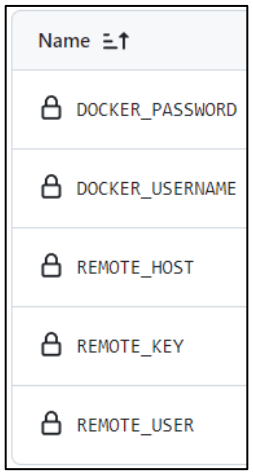
4. Servidores de automatización



Automatizar despliegue con Docker - Trabajo en GitHub

Pasos:

- En el repositorio de GitHub: crear los secretos correspondientes:



Contraseña de tu cuenta de DockerHub

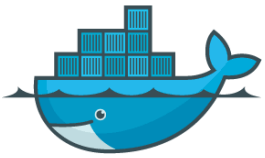
Usuario de tu cuenta de DockerHub

IP de tu instancia EC2 (recuerda que si no la configuras como elástica -fija-, cambiará)

Tu clave PEM (debes copiarla/pegarla exactamente el contenido del fichero)

Usuario de la máquina Ubuntu de EC2 (habitualmente es “ubuntu” por defecto)

4. Servidores de automatización



Automatizar despliegue con Docker - Trabajo en local

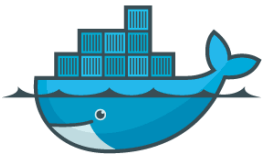
Del tema anterior, `Dockerfile` para la aplicación y `Dockerfile` para la base de datos:

```
/docker_cine
  |-- /mysql
  |      |-- cine.sql
  |      |-- Dockerfile
  |-- /php
  |      |-- Dockerfile
  |      |-- index.php
  |-- compose.yml
```

```
FROM mysql:latest
ENV MYSQL_ROOT_PASSWORD=contrasenyaRoot
ENV MYSQL_DATABASE=cine
ENV MYSQL_USER=usuario1
ENV MYSQL_PASSWORD=contrasenyaUsuario1

COPY cine.sql /docker-entrypoint-initdb.d/
```

```
FROM php:8.2-apache
# Instala y habilita la extensión mysqli
RUN docker-php-ext-install mysqli && docker-php-ext-enable mysqli
COPY . /var/www/html/mi_app_cine
```

4. Servidores de automatización



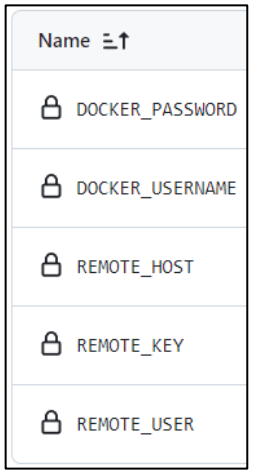
Automatizar despliegue con Docker - Trabajo en local

Del tema anterior, Docker Compose (`compose.yml`) para orquestar los servicios:

```
services:
  db:
    image: rsanzfloridauni/mi_cine:v1
    container_name: contenedorMiCine
    ports:
      - "3306:3306"

  php:
    image: rsanzfloridauni/mi_app_cine:v1
    container_name: contenedorMiApp
    ports:
      - "8080:80"
    depends_on:
      - db

 phpmyadmin:
    image: phpmyadmin/phpmyadmin:latest
    container_name: contenedorPhpMyAdmin
    environment:
      PMA_HOST: db
      MYSQL_ROOT_PASSWORD: contrasenyaRoot
    ports:
      - "8081:80"
    depends_on:
      - db
```

Fichero `.github/workflows/deploy.yml`

```yaml
name: Deploy to Remote Server

on:
  push:
    branches:
      - main

jobs:
  build-and-deploy:
    runs-on: ubuntu-latest

    steps:
      # Clonar el repositorio
      - name: Checkout code
        uses: actions/checkout@v3

      # Iniciar sesión en DockerHub
      - name: Log in to DockerHub
        uses: docker/login-action@v2
        with:
          username: ${{ secrets.DOCKER_USERNAME }}
          password: ${{ secrets.DOCKER_PASSWORD }}

      # Construir y subir la imagen de la base de datos
      - name: Build and push DB Docker image
        run: |
          cd mysql
          docker build -t ${{ secrets.DOCKER_USERNAME }}/mi_cine:v1 .
          docker push ${{ secrets.DOCKER_USERNAME }}/mi_cine:v1

      # Construir y subir la imagen de la aplicación
      - name: Build and push PHP Docker image
        run: |
          cd php
          docker build -t ${{ secrets.DOCKER_USERNAME }}/mi_app_cine:v1 .
          docker push ${{ secrets.DOCKER_USERNAME }}/mi_app_cine:v1

      # Transferir el archivo compose.yml al servidor remoto
      - name: Transfer compose.yml to remote server
        uses: appleboy/scp-action@master
        with:
          host: ${{ secrets.REMOTE_HOST }}
          username: ${{ secrets.REMOTE_USER }}
          key: ${{ secrets.REMOTE_KEY }}
          source: ./compose.yml
          target: ~/deploy/

      # Desplegar la aplicación en el servidor remoto usando docker-compose
      - name: Deploy with Docker Compose
        uses: appleboy/ssh-action@master
        with:
          host: ${{ secrets.REMOTE_HOST }}
          username: ${{ secrets.REMOTE_USER }}
          key: ${{ secrets.REMOTE_KEY }}
          script: |
            cd ~/deploy
            sudo chmod 666 /var/run/docker.sock
            docker compose down || true
            docker compose pull
            docker compose up -d
```



Recuerda definir correctamente los secretos en GitHub

```yaml
name: Deploy to Remote Server

on:
  push:
    branches:
      - main

jobs:
  build-and-deploy:
    runs-on: ubuntu-latest

    steps:
      # Clonar el repositorio
      - name: Checkout code
        uses: actions/checkout@v3

      # Iniciar sesión en DockerHub
      - name: Log in to DockerHub
        uses: docker/login-action@v2
        with:
          username: ${{ secrets.DOCKER_USERNAME }}
          password: ${{ secrets.DOCKER_PASSWORD }}

      # Construir y subir la imagen de la base de datos
      - name: Build and push DB Docker image
        run: |
          cd mysql
          docker build -t ${{ secrets.DOCKER_USERNAME }}/mi_cine:v1 .
          docker push ${{ secrets.DOCKER_USERNAME }}/mi_cine:v1

      # Construir y subir la imagen de la aplicación
      - name: Build and push PHP Docker image
        run: |
          cd php
          docker build -t ${{ secrets.DOCKER_USERNAME }}/mi_app_cine:v1 .
          docker push ${{ secrets.DOCKER_USERNAME }}/mi_app_cine:v1

      # Transferir el archivo compose.yml al servidor remoto
      - name: Transfer compose.yml to remote server
        uses: appleboy/scp-action@master
        with:
          host: ${{ secrets.REMOTE_HOST }}
          username: ${{ secrets.REMOTE_USER }}
          key: ${{ secrets.REMOTE_KEY }}
          source: ./compose.yml
          target: ~/deploy/

      # Desplegar la aplicación en el servidor remoto usando docker-compose
      - name: Deploy with Docker Compose
        uses: appleboy/ssh-action@master
        with:
          host: ${{ secrets.REMOTE_HOST }}
          username: ${{ secrets.REMOTE_USER }}
          key: ${{ secrets.REMOTE_KEY }}
          script: |
            cd ~/deploy
            sudo chmod 666 /var/run/docker.sock
            docker compose down || true
            docker compose pull
            docker compose up -d
```

Define el nombre del flujo de trabajo como "Deploy to Remote Server".

Especifica que el flujo de trabajo se activará automáticamente cuando haya un `push` a la rama `main` (o `master`, si fuera el caso) del repositorio.

Define un trabajo llamado `build-and-deploy` que se ejecutará como parte del flujo de trabajo.
Especifica que este trabajo se ejecutará en un entorno virtual basado en Ubuntu (el más reciente disponible).

Clona el código del repositorio actual en el entorno de trabajo.

Inicia sesión en DockerHub usando las credenciales almacenadas como secretos (`DOCKER_USERNAME` y `DOCKER_PASSWORD`).

Construye una imagen Docker de la base de datos desde el directorio `mysql` y la sube a DockerHub con el nombre `mi_cine:v1`.

Construye una imagen Docker de la aplicación PHP desde el directorio `php` y la sube a DockerHub con el nombre `mi_app_cine:v1`.

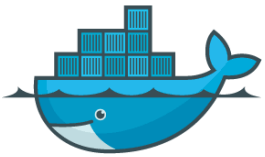
Usa la acción `scp-action` para transferir el archivo `compose.yml` al servidor remoto EC2.

- Se conecta al servidor utilizando las credenciales (`REMOTE_HOST`, `REMOTE_USER` y `REMOTE_KEY`).
- Copia el archivo desde la máquina local a la ubicación remota `~/deploy/`.

Usa la acción `ssh-action` para ejecutar comandos remotos en el servidor:

- Navega al directorio `~/deploy`.
- Cambia permisos del socket de Docker (`/var/run/docker.sock`) para evitar problemas de permisos.
- Baja los contenedores existentes si están corriendo (`docker compose down`).
- Descarga las últimas versiones de las imágenes (`docker compose pull`).
- Levanta los contenedores en modo desacoplado (`docker compose up -d`).

4. Servidores de automatización

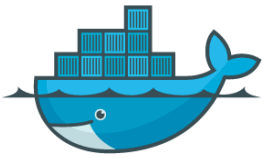


Automatizar despliegue con Docker - Trabajo en local ←→ GitHub

Queda crear el repositorio de Git local (si no lo has hecho ya), vincularlo al remoto y realizar la acción de push.

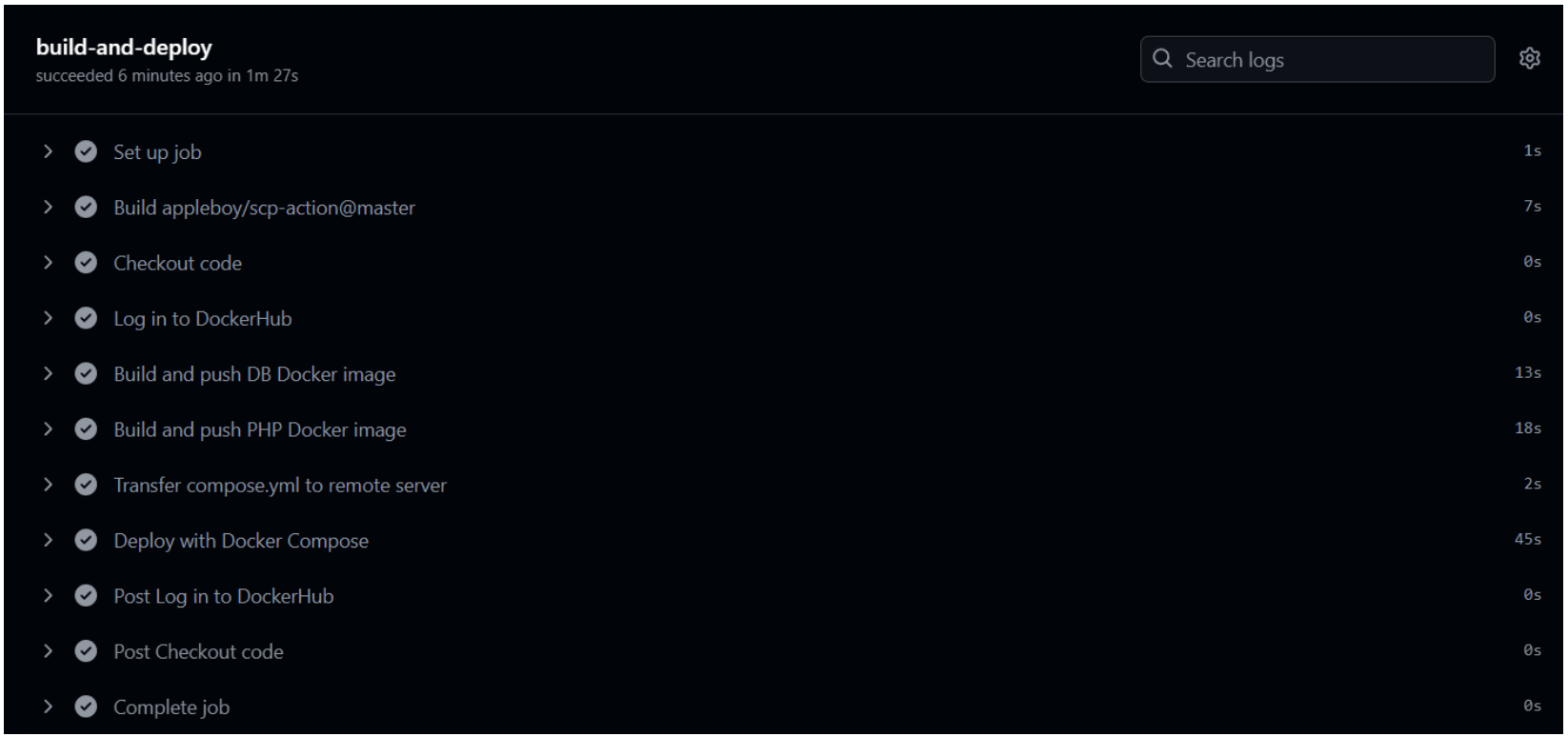
```
git init

git remote add origin https://github.com/tuGitHub/nombreDelRepo.git

git add .

git commit -m “Primer commit”

git push -u origin main
```

4. Servidores de automatización



Automatizar despliegue con Docker - Trabajo en GitHub

Al realizar la acción de `push` desde el repositorio local se desencadena la acción en GitHub Actions. Y si todo funciona correctamente…

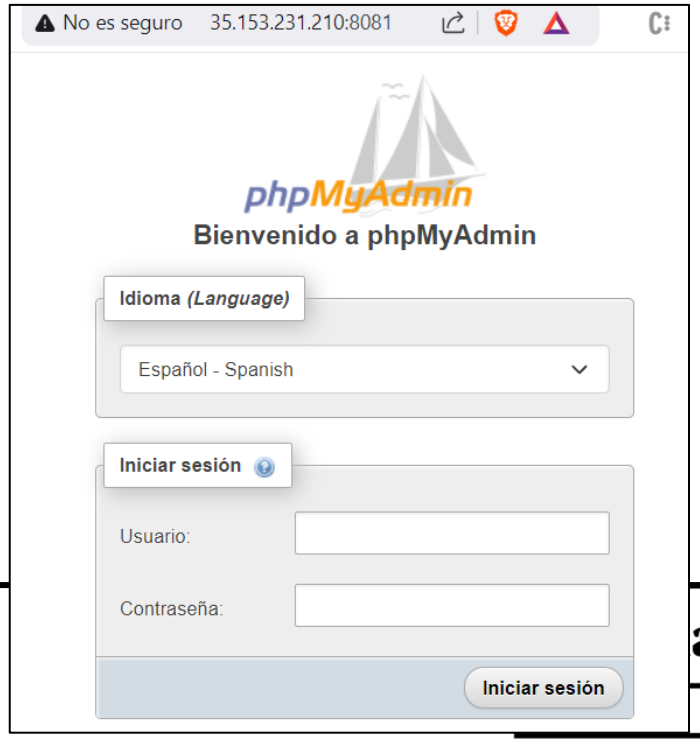


4. Servidores de automatización

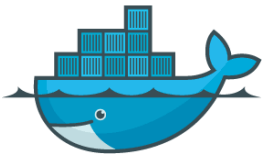
Automatizar despliegue con Docker - Trabajo en local

Y si accedemos desde el navegador a la URL:

Recuerda que los puertos **8080** y **8081** (o los que quieras utilizar) deben estar abiertos en la configuración de seguridad de la instancia EC2.



4. Servidores de automatización



GitHub Actions

Automatizar despliegue con Docker - Trabajo en local ←→ máquina remota

Puedes acceder desde local vía SSH (PuTTy) al servidor EC2 de AWS y comprobar que las imágenes y contenedores funcionan correctamente en la máquina remota:

```
ubuntu@ip-172-31-52-235:~/deploy$ docker images
REPOSITORY                          TAG       IMAGE ID       CREATED              SIZE
rsanzfloridauni/mi_app_cine         v1        3ebc375a8428   About a minute ago   501MB
rsanzfloridauni/mi_cine             v1        3dfd68659c3e   2 minutes ago        604MB
rsanzfloridauni/angular-suma-app    latest    d4d7ffb86093   2 months ago         47.3MB
phpmyadmin/phpmyadmin               latest    933569f3a9f6   17 months ago        562MB
```

```
ubuntu@ip-172-31-52-235:~/deploy$ docker ps
CONTAINER ID   IMAGE                              COMMAND                  CREATED         STATUS         PORTS
          NAMES
77b0975fbba5   phpmyadmin/phpmyadmin:latest       "/docker-entrypoint.…"   9 minutes ago   Up 9 minutes   0.0.0.0:8081->80/tcp, [::]:8081->80/tcp
          contenedorPhpMyAdmin
5ca08b44741a   rsanzfloridauni/mi_app_cine:v1     "docker-php-entrypoi…"   9 minutes ago   Up 9 minutes   0.0.0.0:8080->80/tcp, [::]:8080->80/tcp
          contenedorMiApp
69097db0ba29   rsanzfloridauni/mi_cine:v1         "docker-entrypoint.s…"   9 minutes ago   Up 9 minutes   0.0.0.0:3306->3306/tcp, :::3306->3306/tcp
, 33060/tcp   contenedorMiCine
```

5. Conclusiones

- La integración y despliegue continuos (CI/CD) permiten optimizar la entrega de valor en la cadena de desarrollo y producción de software, automatizando flujos de trabajo que incluyen tareas de desarrollo y operaciones.
- Las partes fundamentales en CI/CD son:
  - Repositorios de código con control de versiones (Git/GitHub)
  - Compilación (*build*) automática (*bundles*/Maven/Make)
  - Implementación de pruebas automatizadas (PHPUnit)
  - Uso de herramientas de automatización (GitHub Actions)
  - Uso de entornos de desarrollo, pruebas y producción (local/MVs/AWS)

6. Anexo: *pipeline* CI/CD para Angular

Vamos a recuperar el ejemplo de la aplicación suma del tema anterior…

El objetivo es replicar el *pipeline/workflow* de GitHub Actions que hemos hecho para la aplicación PHP, pero ahora para una aplicación cliente desarrollada con Angular. Vamos a ver dos *frameworks* de testeo basados en Javascript muy utilizados: Jasmine y JEST.

Jasmine es el *framework* que lleva incorporado Angular por defecto cuando se instala Angular CLI. Es para Javascript, con lo que también se puede utilizar en React u otros *frameworks* de desarrollo cliente de Javascript.

JEST es un *framework* de testeo muy utilizado en Javascript (funciona en proyectos de Angular, Babel, TypesScript, Node, React, Vue, etc.). Hay que instalarlo aparte.

6. Anexo: *pipeline* CI/CD para Angular

Trabajo en local

En el tema anterior se había implementado el componente con los ficheros:

`src/app/suma/suma.component.ts`

`src/app/suma/suma.component.html`

`src/app/app.component.ts`

`src/app/app.component.html`

Para implementar las pruebas hay que editar el fichero `src/app/suma.component.spec.ts` incluyendo las pruebas del componente. En este caso van a ser dos pruebas muy sencillas: la primera es para ver que la suma funciona correctamente y la segunda para ver que el resultado se renderiza en la página correctamente.

```
import { ComponentFixture, TestBed } from '@angular/core/testing';
import { SumaComponent } from './suma.component';

describe('SumaComponent', () => {
  let component: SumaComponent;
  let fixture: ComponentFixture<SumaComponent>;
  beforeEach(async () => {
    await TestBed.configureTestingModule({
      imports: [SumaComponent]
    })
    .compileComponents();
    fixture = TestBed.createComponent(SumaComponent);
    component = fixture.componentInstance;
    fixture.detectChanges();
  });
  it('debe crear', () => {
    expect(component).toBeTruthy();
  });
  it('debe sumar correctamente', () => {
    component.numero1 = 5;
    component.numero2 = 10;
    component.calcularSuma();
    expect(component.resultado).toBe(15);
  });
  it('debe actualizar el resultado en la plantilla HTML', () => {
    component.numero1 = 3;
    component.numero2 = 7;
    component.calcularSuma();
    fixture.detectChanges(); // Trigger change detection
    const compiled = fixture.nativeElement;
    const resultParagraph = compiled.querySelector('p');
    expect(resultParagraph.textContent).toContain('Resultado: 10');
  });
});
```

suma.component.spec.ts

```
import { ComponentFixture, TestBed } from '@angular/core/testing';
import { SumaComponent } from './suma.component';

describe('SumaComponent', () => {
  let component: SumaComponent;
  let fixture: ComponentFixture<SumaComponent>;
  beforeEach(async () => {
    await TestBed.configureTestingModule({
      imports: [SumaComponent]
    })
    .compileComponents();
    fixture = TestBed.createComponent(SumaComponent);
    component = fixture.componentInstance;
    fixture.detectChanges();
  });
  it('debe crear', () => {
    expect(component).toBeTruthy();
  });
  it('debe sumar correctamente', () => {
    component.numero1 = 5;
    component.numero2 = 10;
    component.calcularSuma();
    expect(component.resultado).toBe(15);
  });
  it('debe actualizar el resultado en la plantilla HTML', () => {
    component.numero1 = 3;
    component.numero2 = 7;
    component.calcularSuma();
    fixture.detectChanges(); // Trigger change detection
    const compiled = fixture.nativeElement;
    const resultParagraph = compiled.querySelector('p');
    expect(resultParagraph.textContent).toContain('Resultado: 10');
  });
});
```

Importa componente a testear

Descripción del bloque de pruebas para `SumaComponent`

Declara variable `component` (para facilitar y reutilizar código)

Variable para manejar el componente en el entorno de pruebas

Función que inicializa el entorno de pruebas para cada prueba

Definición de una prueba: comprobar que el componente se ha creado y no es nulo

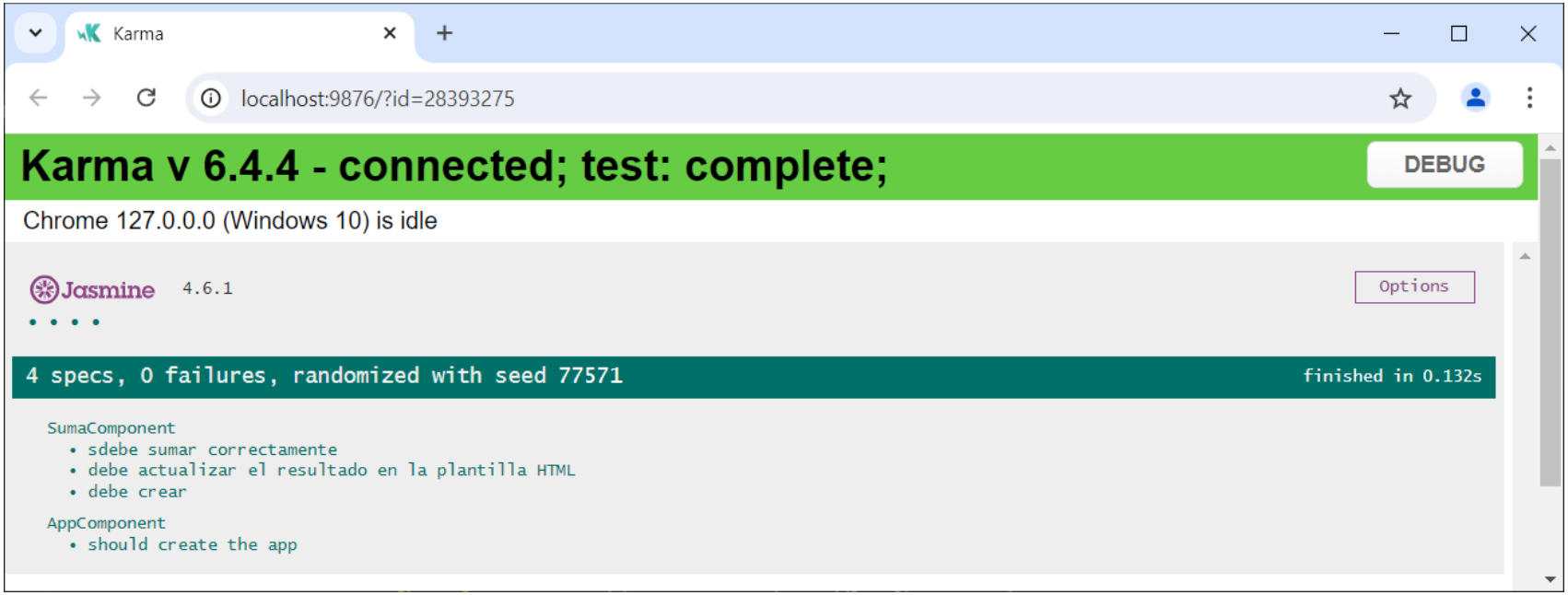
Definición de una prueba: comprobar que la suma se realiza correctamente, definiendo dos números y llamando al método "`calcularSuma()`" del componente. Se utilizan los nombres de los atributos del componente (`numero1`, `numero2` y `resultado`).

Definición de una prueba: comprobar que el resultado de la suma se actualiza correctamente en el DOM (Document Object Modelo) o página web HTML. `fixture.detectChanges()` detecta los cambios y actualiza la vista, `fixture.nativeElement` obtiene el elemento DOM que ha cambiado, concretamente se trata de un elemento de tipo `<p>` (ver `suma.component.html`). Finalmente se verifica que el elemento `<p>` tiene como contenido "Resultado 10" para la suma de los números de esta prueba.

6. Anexo: *pipeline* CI/CD para Angular

Trabajo en local - Testeo con Jasmine/Karma

Ejecutar el test para ver que funcione correctamente desde el terminal: `npm test`



Ejemplo de test correcto

NOTA: El *framework* Jasmine lleva incorporado un servidor web llamado Karma para visualizar el resultado de las pruebas. El servidor se ejecuta tras realizar la prueba.

6. Anexo: *pipeline* CI/CD para Angular

Trabajo en local - Testeo con Jasmine/Karma

Ejecutar el test para ver que funcione correctamente desde el terminal: `npm test`

Ejemplo de test incorrecto (se ha forzado que el Resultado sea 9 en vez de 10)

6. Anexo: *pipeline* CI/CD para Angular

Trabajo en local - Testeo con JEST

Dado que JEST no viene incorporado por defecto en Angular, hay que instalarlo y realizar algunas tareas adicionales de configuración en el proyecto.

1. Instalar dependencias:

```
npm install --save-dev jest jest-preset-angular @types/jest ts-jest
```

2. Añadir al fichero `package.json` la configuración de JEST:

```
"jest": {
  "preset": "jest-preset-angular",
  "setupFilesAfterEnv": ["<rootDir>/setup-jest.ts"],
  "testPathIgnorePatterns": [
    "<rootDir>/node_modules/",
    "<rootDir>/dist/",
    "<rootDir>/e2e/"
  ],
  "globals": {
    "ts-jest": {
      "tsconfig": "<rootDir>/tsconfig.spec.json",
      "stringifyContentPathRegex": "\\.html$",
      "isolatedModules": true
    }
  },
  "moduleNameMapper": {
    "^src/(.*)$": "<rootDir>/src/$1"
  }
}
```

6. Anexo: *pipeline* CI/CD para Angular

Trabajo en local - Testeo con JEST

3. En la raíz del proyecto, crear el fichero `setup-test.js` con el siguiente contenido:

```
import 'jest-preset-angular/setup-jest';
```

4. Modificar el fichero `tsconfig.spec.json` para que incluya el compilador JEST/Node:

```
{
  "extends": "./tsconfig.json",
  "compilerOptions": {
    "outDir": "./out-tsc/spec",
    "types": ["jest", "node"]
  },
  "files": ["src/test.ts"],
  "include": ["src/**/*.spec.ts", "src/**/*.d.ts"]
}
```

5. Modifica la entrada “`test`” del atributo “`scripts`” de `package.json` para que sea “`jest`”:

```
"scripts": {
  "test": "jest",
  ...
}
```

6. Anexo: *pipeline* CI/CD para Angular

Trabajo en local - Testeo con JEST

6. Ejecutar el test: `npm test`

```
 PASS  src/app/app.component.spec.ts (6.876 s)
 PASS  src/app/suma/suma.component.spec.ts (6.923 s)

Test Suites: 2 passed, 2 total
Tests:       4 passed, 4 total
Snapshots:   0 total
Time:        10.107 s
Ran all test suites.
```

Y si forzamos que haya un error… ➔

NOTA: A diferencias del *framework* Jasmine, que lleva incorporado el servidor web Karma para visualizar el resultado de las pruebas, con JEST toda la información de las pruebas aparece en el terminal.

6. Anexo: *pipeline* CI/CD para Angular

Trabajo en local - Testeo con JEST

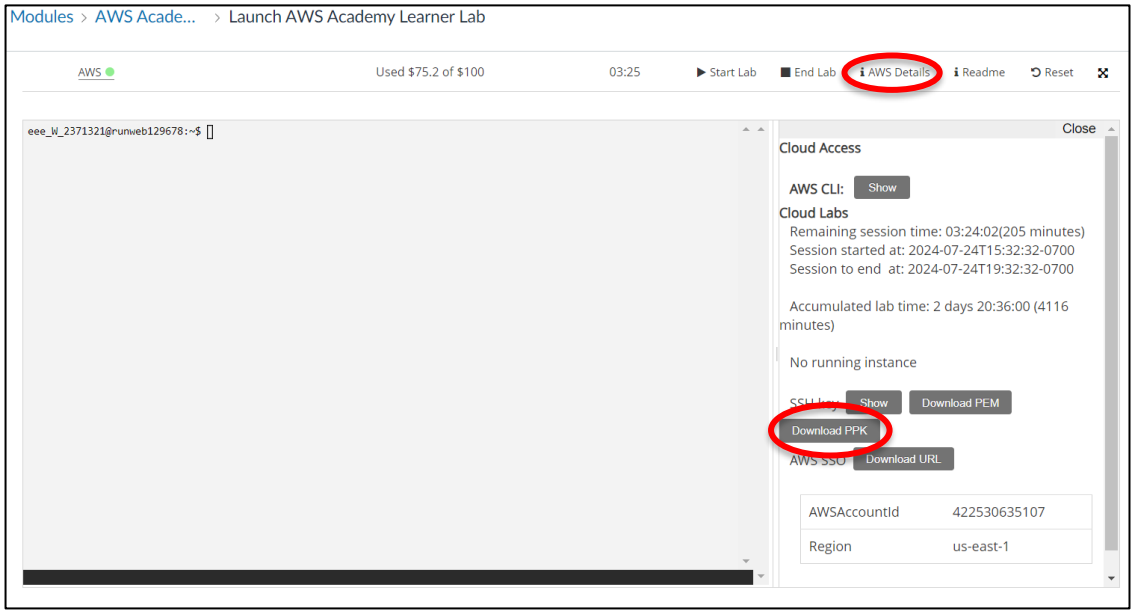
```
PASS src/app/app.component.spec.ts
FAIL src/app/suma/suma.component.spec.ts
  ● SumaComponent › debe actualizar el resultado en la plantilla HTML

    expect(received).toContain(expected) // indexOf

    Expected substring: "Resultado: 9"
    Received string:    "Resultado: 10"

      30 |     const compiled = fixture.nativeElement;
      31 |     const resultParagraph = compiled.querySelector('p');
    > 32 |     expect(resultParagraph.textContent).toContain('Resultado: 9');
    > 32 |     expect(resultParagraph.textContent).toContain('Resultado: 9');
         |                                         ^
      33 |   });
      34 | });
      35 |

      at src/app/suma/suma.component.spec.ts:32:41
      at _ZoneDelegate.invoke (node_modules/zone.js/bundles/zone.umd.js:416:32)
      at ProxyZoneSpec.Object.<anonymous>.ProxyZoneSpec.onInvoke (node_modules/zone.js/bundles/zone-testing.umd.js:2164:43)
      at _ZoneDelegate.invoke (node_modules/zone.js/bundles/zone.umd.js:415:38)
      at ZoneImpl.run (node_modules/zone.js/bundles/zone.umd.js:147:47)
      at Object.wrappedFunc (node_modules/zone.js/bundles/zone-testing.umd.js:450:38)

Test Suites: 1 failed, 1 passed, 2 total
Tests:       1 failed, 3 passed, 4 total
Snapshots:   0 total
Time:        4.578 s, estimated 7 s
Ran all test suites.
```

6. Anexo: *pipeline* CI/CD para Angular

Preparación del servidor en AWS

1. Si no la has creado con anterioridad, crea una instancia EC2 (máquina virtual) de AWS con Ubuntu (puedes seguir la guía que hay disponible en Florida Oberta).

2. Desde la página de AWS Academy Learner Lab, en AWS Details, descarga la clave SSH con formato PPK (la necesitarás para acceder por PuTTy ➔ ver tema 2 ➔).



6. Anexo: *pipeline* CI/CD para Angular

Preparación del servidor en AWS

3. Accede por SSH a la instancia EC2 (ver **tema 2**)
4. Actualiza la lista de paquetes: `sudo apt update`
5. Instala Nginx: `sudo apt install nginx`
6. Comprueba que funciona poniendo en el navegador: http://IP_EC2_AWS
7. El directorio por defecto donde se alojan las aplicaciones es `/var/www/html`. Vamos a crear un directorio específico para nuestro proyecto `/var/www/html/suma-angular`. Accede a `./html` y crea el directorio: `sudo mkdir suma-angular`

6. Anexo: *pipeline* CI/CD para Angular

Trabajo en GitHub

1. Crea un repositorio público en GitHub (es necesario que sea público para el plan gratuito de GitHub Actions). En este ejemplo lo he llamado `suma-angular`.

2. En “Settings”, accede a “Secrets and variables → Actions”. Este apartado permite almacenar de manera segura variables con información sensible, como nombres de usuario, contraseñas, IPs, etc. Dentro de “Repository secrets”, vamos a crear los siguientes:

   Name: `DEPLOY_PATH` **Secret**: ruta de despliegue (p.ej. `/var/www/html/suma-app`)
   Name: `HOST` **Secret**: IP de la instancia EC2 de AWS*
   Name: `SSH_PRIVATE_KEY` **Secret**: contenido de la clave .pem de AWS**
   Name: `USERNAME` **Secret**: nombre de usuario (p.ej. `ubuntu`)

* Ten en cuenta que si no configuras la IP pública de tu EC2 como elástica (“fija”), cada vez que inicies la EC2 cambiará, con lo que tendrás que cambiar también este secreto. Por tanto, es aconsejable configurar siempre tu máquina con IP elástica para evitar errores.

** No confundir la clave .pem con la clave .ppk, que se utiliza solo para PuTTy.

6. Anexo: *pipeline* CI/CD para Angular

Trabajo en Local ←→ GitHub

1. Crea un fichero `.gitignore` para que no se sincronicen dependencias y otros ficheros que no es necesario tener en GitHub (ver diapositiva siguiente).
2. En el directorio local del proyecto crea el fichero YAML (crea también la ruta) `./.github/workflows/despliegue.yml` (ver diapositiva siguiente).
3. Iniciar el repositorio: `git init`
4. Añadir el remoto: `git remote add origin https://github.com/rsanzfloridauni/suma-angular.git`
5. Añade todos los ficheros para preparar el commit: `git add .`
6. Haz el *commit*: `git commit -m “Primer commit”`
7. Haz el *push* a GitHub: `git push -u origin main`

Si el *push* se realiza correctamente, se enviará todo al repositorio remoto y automáticamente se desencadenará la “Action” programada `despliegue.yml`. En GitHub, accede a la pestaña “Actions” del repositorio. Si lo haces inmediatamente después de hacer el *push*, te dará a tiempo a ver que la “Action” está aún en proceso.

.gitignore

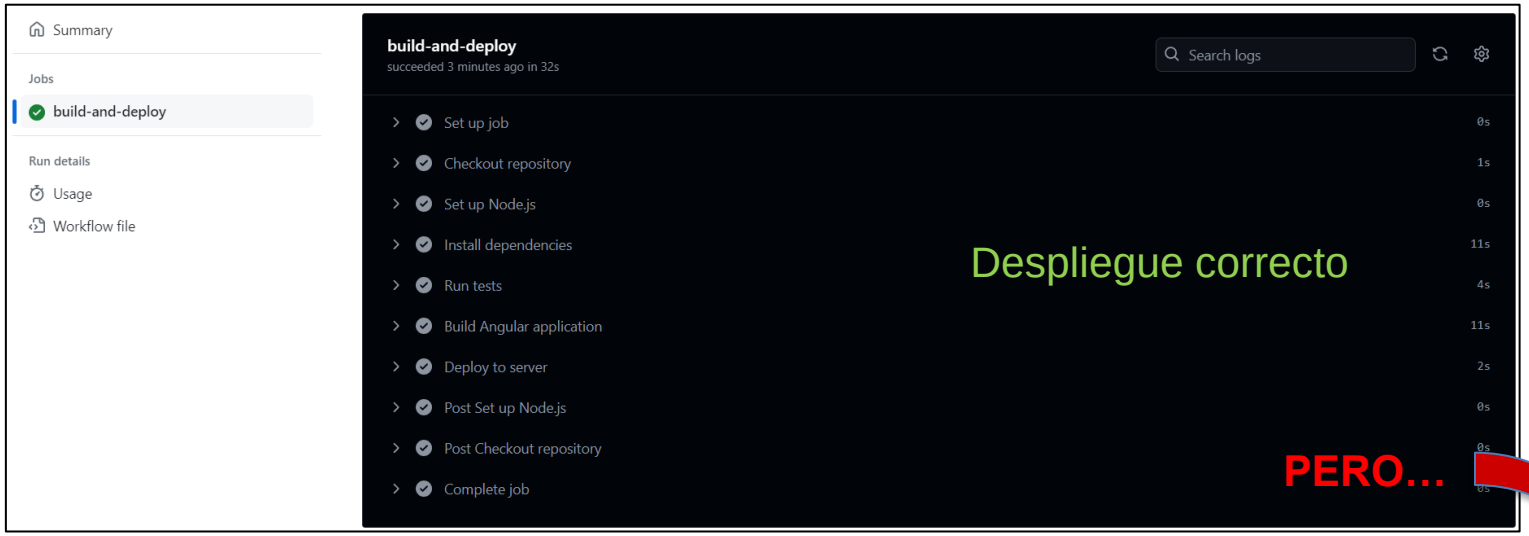
```
# Compiled output
/dist
/tmp
/out-tsc
/bazel-out

# Node
/node_modules
npm-debug.log
yarn-error.log

# IDEs and editors
.idea/
.project
.classpath
.c9/
*.launch
.settings/
*.sublime-workspace

# Visual Studio Code
.vscode/*
!.vscode/settings.json
!.vscode/tasks.json
!.vscode/launch.json
!.vscode/extensions.json
.history/*

# Miscellaneous
/.angular/cache
.sass-cache/
/connect.lock
/coverage
/libpeerconnection.log
testem.log
/typings

# System files
.DS_Store
Thumbs.db
```

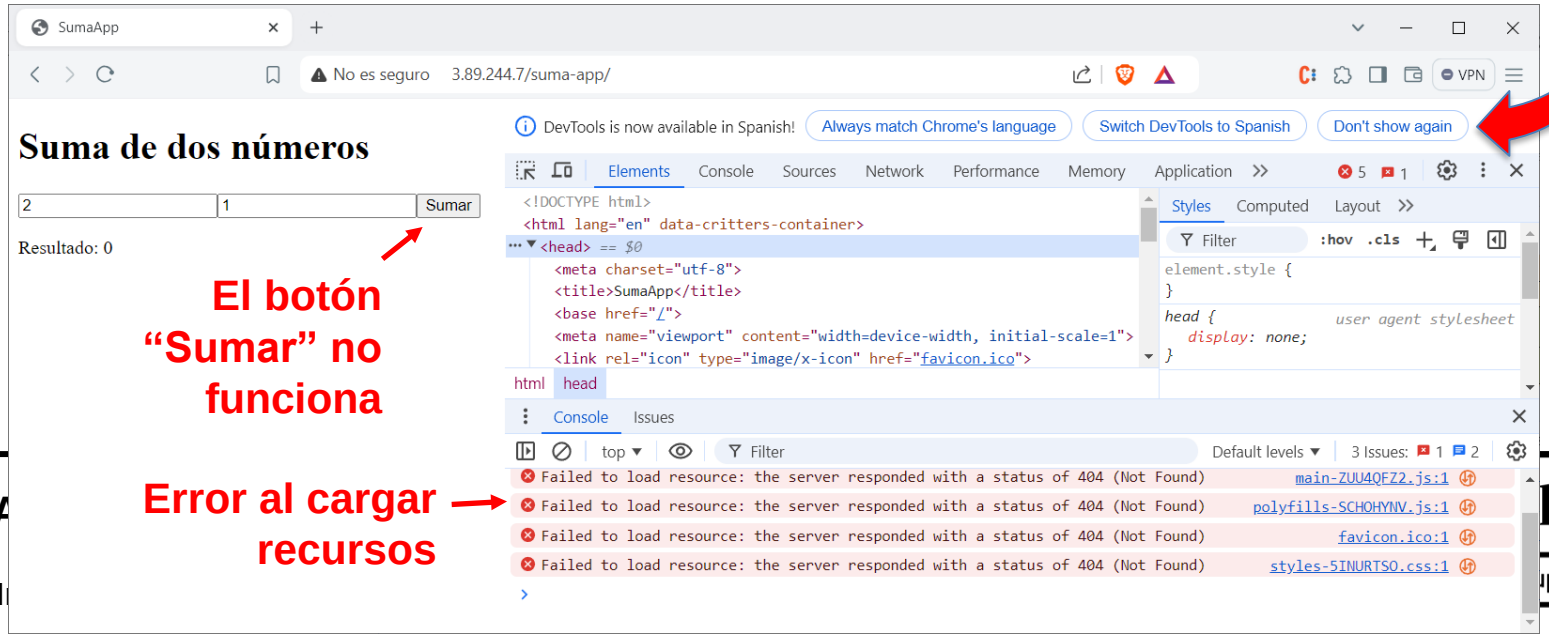
despliegue.yml

```
name: Deploy to Server
on:
  push:
    branches:
      - main  # o cualquier rama en la que desees activar este pipeline
jobs:
  build-and-deploy:
    runs-on: ubuntu-latest
    steps:
    - name: Checkout repository
      uses: actions/checkout@v2

    - name: Set up Node.js
      uses: actions/setup-node@v2
      with:
        node-version: '20'  # version LTS

    - name: Install dependencies
      run: npm install

    - name: Run tests
      run: npm test

    - name: Build Angular application
      run: npm run build

    - name: Deploy to server
      env:
        SSH_PRIVATE_KEY: ${{ secrets.SSH_PRIVATE_KEY }}
        HOST: ${{ secrets.HOST }}
        USERNAME: ${{ secrets.USERNAME }}
        DEPLOY_PATH: ${{ secrets.DEPLOY_PATH }}
      run: |
        echo "Deploying to server..."
        mkdir -p ~/.ssh
        echo "$SSH_PRIVATE_KEY" | tr -d '\r' > ~/.ssh/id_rsa
        chmod 600 ~/.ssh/id_rsa
        ssh -o StrictHostKeyChecking=no $USERNAME@$HOST "mkdir -p $DEPLOY_PATH“
        ssh -i ~/.ssh/id_rsa $DEPLOY_USER@$DEPLOY_SERVER "rm -rf $DEPLOY_PATH/*"
        rsync -avz --delete ./dist/suma-app/browser/ $USERNAME@$HOST:$DEPLOY_PATH
        ssh $USERNAME@$HOST 'sudo systemctl restart nginx'
```

6. Anexo: *pipeline* CI/CD para Angular

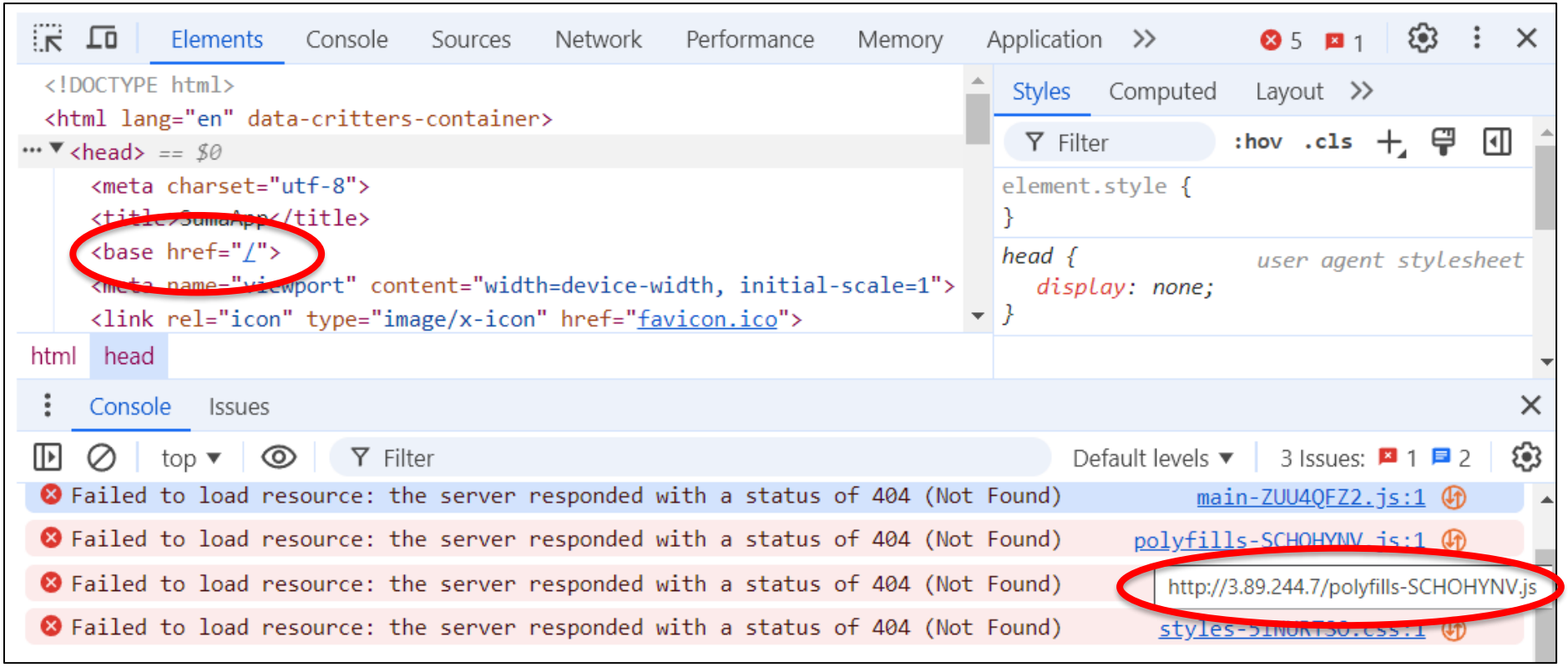
GitHub Action finalizada y despliegue realizado





6. Anexo: *pipeline* CI/CD para Angular

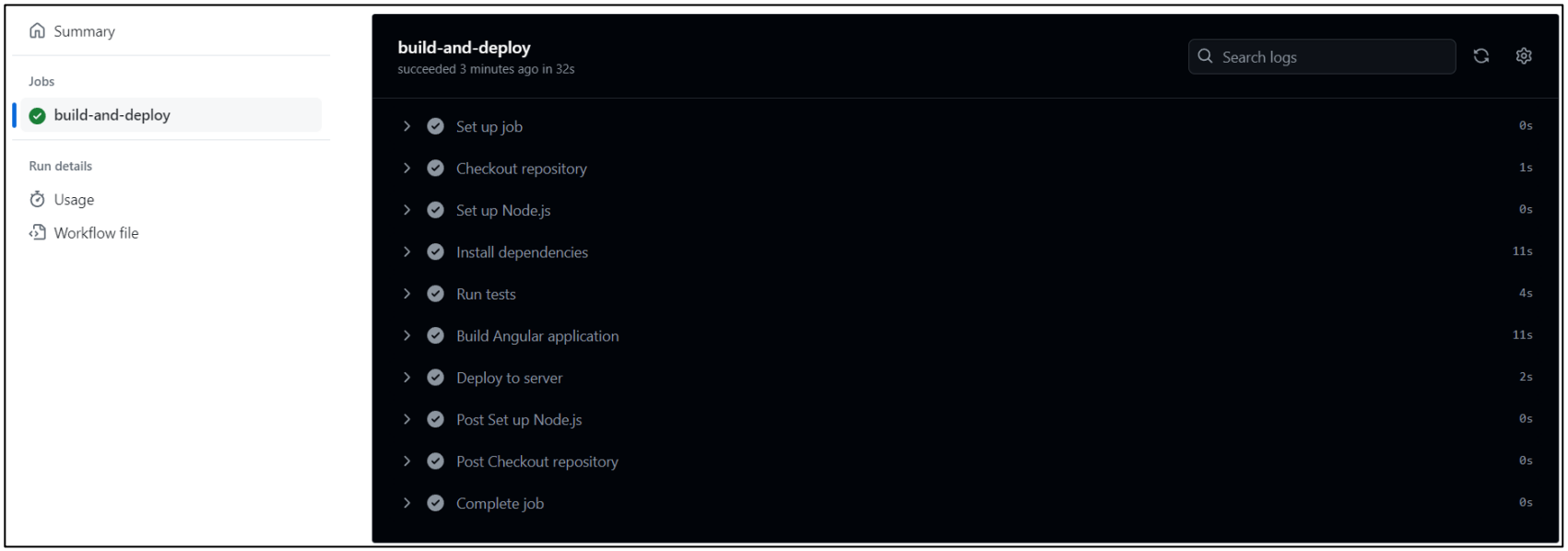
GitHub Action finalizada y despliegue realizado

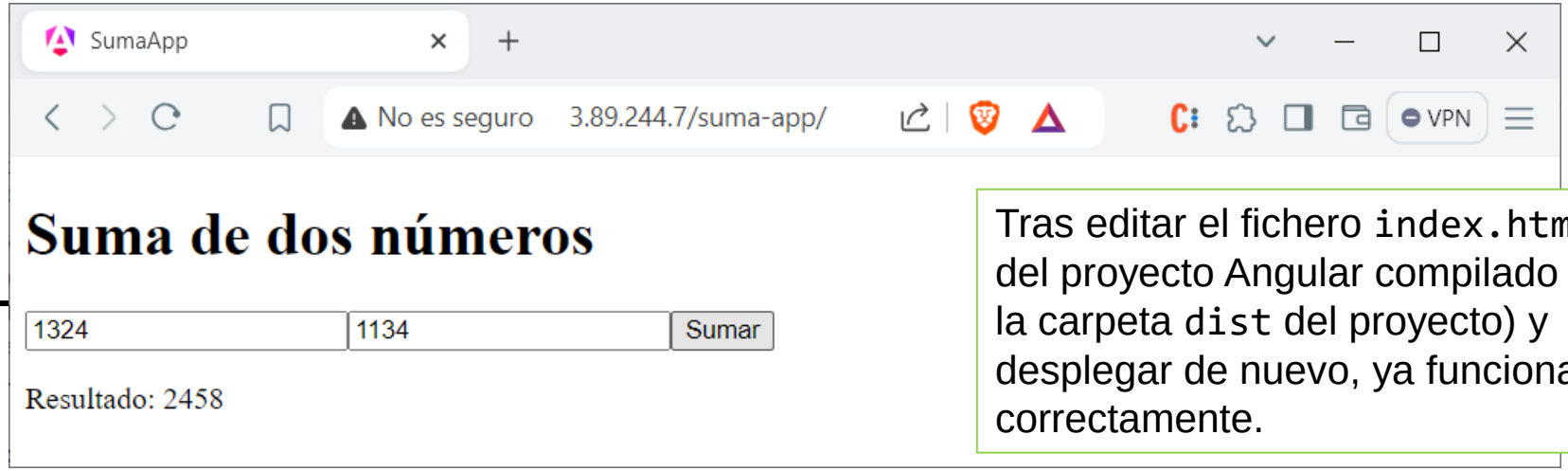


Hemos desplegado nuestra aplicación en http://IP/suma-app, pero está buscando los recursos en http://IP/. El fichero `index.html` que crea Angular por defecto no tiene en cuenta que se puede desplegar la aplicación en un subdirectorio del servidor, por lo que hay que indicarlo manualmente. Basta editar el campo `<base href="/">` y cambiarlo por `<base href="/suma-app/">` (o el que hayas definido en la ruta de despliegue). Este campo `href` debe coincidir con el subdirectorio del servidor Nginx (o Apache) (ruta `/var/www/html/subdirectorio`).

6. Anexo: *pipeline* CI/CD para Angular

GitHub Action finalizada y despliegue realizado





Tras editar el fichero `index.html` del proyecto Angular compilado (en la carpeta `dist` del proyecto) y desplegar de nuevo, ya funciona correctamente.

6. Anexo: *pipeline* CI/CD para Angular

Sugerencia…

Por último, puedes completar el *pipeline* creando de manera automática una imagen personalizada de Docker con tu aplicación y subiéndola a tu cuenta de Docker Hub, tal y como hemos hecho para el caso de la aplicación Hola Mundo Multilenguaje de PHP.

Puedes partir del fichero `Dockerfile` para una aplicación Angular que desarrollamos en el tema anterior.