Icono *Training Consulting*

Formación

Testing Front End

2025

Formador

Ana Isabel Vegas

INGENIERA INFORMÁTICA con Master Máster Universitario en Gestión y Análisis de Grandes Volúmenes de Datos: Big Data, tiene la certificación PCEP en Lenguaje de Programación Python y la certificación JSE en Javascript. Además de las certificaciones SCJP Sun Certified Programmer for the Java 2 Platform Standard Edition, SCWD Sun Certified Web Component Developer for J2EE 5, SCBCD Sun Certified Business Component Developer for J2EE 5, SCEA Sun Certified Enterprise Architect for J2EE 5.

Desarrolladora de Aplicaciones FULLSTACK, se dedica desde hace + de 20 años a la CONSULTORÍA y FORMACIÓN en tecnologías del área de DESARROLLO y PROGRAMACIÓN.

training@iconotc.com

Testing Front End

❑ **Duración:** 20 horas

❑ **Modalidad:** On-line

❑ **Fechas/Horario:**

- Días 20, 21, 22, 23, 27, 28, 29 y 30 Octubre 2025
- Horario 15:00 – 17:30 hs.

❑ **Contenidos:**

- Introducción
- Pruebas de Unidad e Integración en JavaScript
- Pruebas de Aceptación
- Testing Library
- Pruebas de Accesibilidad
- Integración DevOps

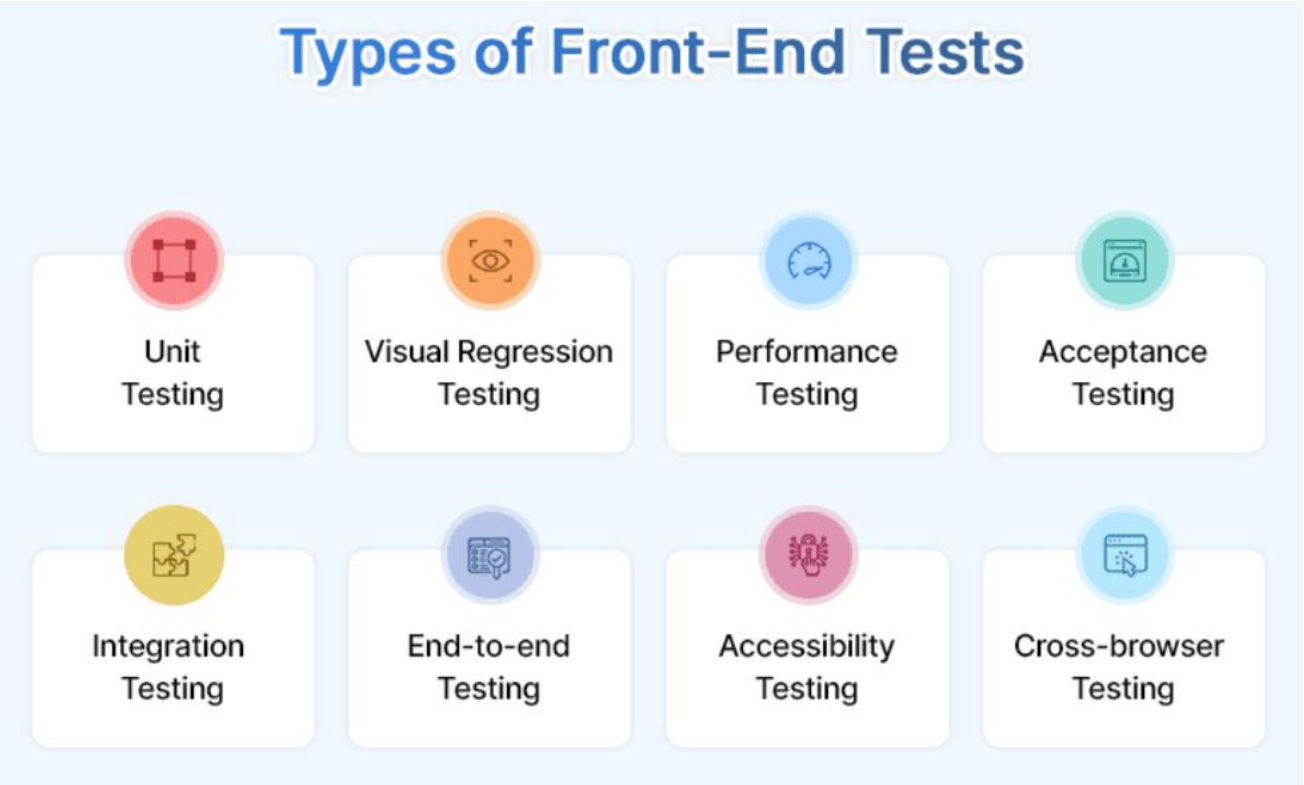
Introducción

Tema 1

Testing Front End

- El testing front-end es una parte esencial del desarrollo moderno de aplicaciones web.
- Su objetivo es garantizar que las interfaces que los usuarios utilizan funcionen correctamente, sean accesibles y mantengan una experiencia de uso fluida ante cualquier cambio en el código.
- En este curso aprenderás los principios y las herramientas más utilizadas para automatizar las pruebas en el lado del cliente, detectando errores antes de que lleguen a producción.

Tipos de pruebas



Pruebas esenciales

- **Pruebas unitarias**: Verifican pequeñas unidades de código, como funciones, métodos o componentes individuales, de forma aislada para asegurar que funcionan correctamente por sí solas.
- **Pruebas de integración**: Evalúan cómo interactúan varios componentes o módulos al trabajar juntos, asegurando que sus conexiones y colaboraciones sean correctas.
- **Pruebas funcionales/end-to-end (E2E)**: Simulan el comportamiento real del usuario y prueban flujos completos dentro de la aplicación, asegurando que todo el sistema funcione según lo esperado desde la perspectiva del usuario final.
- **Pruebas de regresión**: Comprueban que los cambios recientes en el código no hayan roto funcionalidades que antes funcionaban bien.
- **Pruebas visuales**: Detectan cambios inesperados en la interfaz gráfica comparando capturas de pantalla antes y después de una modificación.

Otras pruebas importantes

- **Pruebas entre navegadores (cross-browser):** Garantizan que la aplicación funcione correctamente en distintos navegadores, sistemas operativos y dispositivos.
- **Pruebas de rendimiento:** Evalúan la velocidad, capacidad de respuesta y estabilidad de la aplicación bajo diferentes condiciones de uso.
- P**ruebas de accesibilidad:** Se utilizan para evaluar y optimizar la interfaz de usuario de una aplicación web para garantizar que sea utilizable por personas con diferentes discapacidades, ya sean visuales, auditivas, motoras o cognitivas.

Pruebas de Unidad e Integración en JavaScript

Tema 2

Pruebas unitarias y de integración

- Las pruebas unitarias en JavaScript consisten en verificar que una unidad individual del código (como una función o método) funcione correctamente aislada del resto del sistema. Por ejemplo, se prueba que una función suma correctamente como unidad independiente.

- Las pruebas de integración van un paso más allá, asegurando que varias unidades de código funcionen correctamente juntas, verificando la interacción entre componentes.

Frameworks utilizados

- Para realizar tanto pruebas unitarias como de integración en JavaScript, se utilizan comúnmente frameworks como Jest, Mocha, Jasmine, Karma, entre otros. Jest es uno de los más populares y fáciles de usar, proporcionando una experiencia de "configuración cero" y soporte para pruebas unitarias y de integración. Mocha es otra opción flexible especialmente cuando se necesita escribir pruebas asíncronas. Estos frameworks ofrecen herramientas para escribir pruebas, hacer aserciones, simular dependencias y automatizar la ejecución de pruebas.

Herramientas principales

- **Jest**: Popular, usado por grandes empresas, soporta pruebas unitarias y de integración, fácil configuración.
- **Mocha**: Permite pruebas asíncronas, muy flexible.
- **Jasmine**: Sin necesidad de DOM, sintaxis sencilla.
- **Karma**: Entorno para ejecutar pruebas con varios frameworks.
- **QUnit**: Versátil para cliente y servidor.
- **Unit.js**: Biblioteca de aserciones compatible con otros frameworks.

Jest

- Jest es un framework de pruebas JavaScript desarrollado por Facebook, muy popular por su facilidad para escribir y correr pruebas, sin necesidad de configuraciones complicadas. Sirve para hacer pruebas unitarias, de integración y más.

sum.js

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

module.exports = suma;
```

test.sum.js

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

test('suma 1 + 2 es igual a 3', () => {
  expect(suma(1, 2)).toBe(3);
});
```

Sintaxis de Jest

- La sintaxis de Jest es intuitiva y está diseñada para facilitar la escritura de pruebas en JavaScript. Jest usa funciones globales principales para organizar, escribir y validar pruebas:
  - **describe(nombre, fn)**: Agrupa pruebas relacionadas dentro de un bloque con una descripción clara, útil para organizar el código de test.
  - **test(nombre, fn) o su alias it(nombre, fn):** Define un caso de prueba específico con su descripción y la función donde se ejecutan las aserciones.
  - **expect(valor):** Función que crea una expectativa sobre un valor, a la que se le aplican "matchers" para hacer aserciones.

Matchers comunes en Jest

- La sintaxis de Jest es intuitiva y está diseñada para facilitar la escritura de pruebas en JavaScript. Jest usa funciones globales principales para organizar, escribir y validar pruebas:
    - **toBe(valor):** Verifica igualdad estricta (===).
    - **toEqual(valor):** Verifica igualdad profunda de objetos o arrays.
    - **toBeTruthy():** Confirma que el valor es "truthy".
    - **toBeFalsy():** Confirma que el valor es "falsy".
    - **toContain(elemento):** Verifica que un array o string contiene un elemento.
    - **toThrow():** Usa para verificar que una función lance un error.

Hocks para configurar pruebas en Jest

- Estos hooks permiten preparar o limpiar estados antes y después de las pruebas, por ejemplo, inicializar datos o mocks:
    - **beforeEach(fn):** Código que se ejecuta antes de cada prueba.
    - **afterEach(fn):** Código que se ejecuta después de cada prueba.
    - **beforeAll(fn):** Se ejecuta una vez antes de todas las pruebas.
    - **afterAll(fn):** Se ejecuta una vez después de todas las pruebas.

Pruebas asíncronas con Jest

- Jest maneja pruebas asíncronas con funciones async/await, devolviendo Promesas, o usando callbacks con done().

```
test('prueba asíncrona con async/await', async () => {
  const data = await fetchData();
  expect(data).toBe('valor esperado');
});
```

Mocha

- Mocha es un framework de testing para JavaScript que se puede usar tanto en Node.js como en navegadores.
- Es muy flexible, permite escribir pruebas unitarias y de integración, soporta pruebas asíncronas y se puede combinar con librerías de aserción como Chai.
- Es una herramienta popular para asegurar la calidad del código en proyectos JavaScript.

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

module.exports = suma;  // Exportamos con CommonJS
```

```
const expect = require('chai').expect;
const suma = require('../sum');  // Importamos con require

describe('Pruebas de suma', function() {
  it('debería devolver 3 al sumar 1 + 2', function() {
    expect(suma(1, 2)).to.equal(3);
  });
});
```

Jasmine

- Jasmine es un framework de pruebas unitarias para JavaScript ampliamente usado, especialmente en proyectos Angular, aunque es independiente del framework. Su objetivo es facilitar la escritura de tests con una sintaxis clara y legible que también sirve como documentación del comportamiento del código.

- Características principales:

  - No depende de otros frameworks ni del DOM, funciona en varios entornos.
  - Sigue el paradigma Behavior-Driven Development (BDD), permitiendo definir suites de pruebas con describe y casos con it.
  - Ofrece funciones para configuraciones previas (beforeEach, beforeAll) y de limpieza posterior (afterEach, afterAll).
  - Usa expect() junto con “matchers” para validar resultados, como toBe(), toEqual(), toBeTruthy(), entre otros.
  - Permite espiar funciones con spyOn() para verificar interacciones.

Ejemplo de Jasmine

```
describe('Calculadora', () => {
  let total;

  beforeEach(() => {
    total = 0;
  });

  it('suma correctamente dos números', () => {
    total = 1 + 2;
    expect(total).toBe(3);
  });

  it('resta correctamente', () => {
    total = 5 - 3;
    expect(total).toBe(2);
  });
});
```

Karma

- Karma es un *test runner* para JavaScript cuya función principal es ejecutar pruebas en diferentes navegadores de manera automática, facilitando la validación continua del código. No es un framework de pruebas por sí mismo, sino que se integra con otros frameworks como Jasmine, Mocha o QUnit para correr las pruebas escritas con ellos.

- Características:

  - Ejecuta pruebas en múltiples navegadores (Chrome, Firefox, Safari, IE, etc.).
  - Facilita la integración con entornos de desarrollo y sistemas de integración continua (CI/CD).
  - Detecta cambios en archivos y vuelve a lanzar las pruebas automáticamente.
  - Permite configurar reportes y métricas de pruebas.
  - Se utiliza comúnmente junto con Jasmine para testing en Angular, donde viene preconfigurado en Angular CLI.

Flujo básico con karma

1. Se escribe la prueba usando un framework (por ejemplo Jasmine).
2. Karma carga los archivos y ejecuta las pruebas en los navegadores configurados.
3. Muestra los resultados en la consola o navegadores en tiempo real.
4. Continúa escuchando cambios para ejecución continua.

Como iniciar karma

- Instalar paquetes: karma, karma-cli, karma-jasmine, karma-chrome-launcher, etc.
- Configurar con karma.conf.js generado por karma init.
- Ejecutar pruebas con karma start o ng test en Angular.
- Configurar pruebas headless para entorno servidor o CI.

Pruebas de aceptación

Tema 3

Pruebas de aceptación

- Las pruebas de aceptación son un tipo de prueba de software cuyo objetivo es validar que un sistema cumple con los requisitos y expectativas del cliente o usuario final antes de su lanzamiento. Son la última etapa formal del proceso de pruebas y permiten al usuario confirmar que el producto satisface su necesidad y está listo para su uso real.

- Características principales de las pruebas de aceptación:

    - Se basan en criterios de aceptación claros y medibles definidos desde el inicio del proyecto.
    - Validan funcionalidades, rendimiento, usabilidad y cumplimiento de requisitos legales o contractuales.
    - Pueden ser realizadas por usuarios finales, clientes o representantes en colaboración con el equipo de desarrollo.
    - Incluyen escenarios del mundo real o casos de uso típicos.

Principales frameworks para pruebas de aceptación

- **Cucumber**: Muy popular para pruebas basadas en BDD (Behavior Driven Development). Permite escribir pruebas en lenguaje natural (Gherkin), fáciles de entender para perfiles no técnicos. Ideal para validar requisitos y escenarios de usuario.

- **Selenium** es una herramienta ampliamente utilizada para pruebas de aceptación, especialmente para aplicaciones web. Su función principal es automatizar la interacción con navegadores, simulando el comportamiento de un usuario real: hacer clic, llenar formularios, navegar, validar elementos, etc.

- **Cypress**: Herramienta moderna para pruebas de extremo a extremo en aplicaciones web, también usada en validaciones de aceptación gracias a su velocidad y facilidad de uso.

- **Serenity BDD**: Construido sobre Selenium y Cucumber, mejora la gestión de pruebas de aceptación ofreciendo mejores reportes, documentación automática y soporte para pruebas de integración y aceptación.

Gherkin

- Gherkin es un lenguaje sencillo, utilizado en testing para describir escenarios de prueba de manera que cualquier persona (técnica o no técnica) los pueda entender. Es el estándar para herramientas BDD como Cucumber y permite definir comportamientos esperados del sistema.

- Elementos clave de Gherkin
  - **Feature**: Describe la funcionalidad o conjunto de pruebas que se van a validar.
  - **Scenario**: Es el caso de prueba individual, que representa un comportamiento específico.
  - **Given**: Describe el estado inicial necesario (Dado).
  - **When**: Indica la acción que realiza el usuario (Cuando).
  - **Then**: Representa el resultado esperado tras la acción (Entonces).
  - **And/But**: Permite encadenar varios pasos con conectores (Y/Pero).

Gherkin

- Buenas prácticas:
  - Un escenario debe ser breve y fácil de entender.
  - Usa el patrón Given-When-Then siempre en ese orden.
  - Si tienes acciones o resultados adicionales, usa And y But para clarificar.
  - Escribe los escenarios como si fueras el usuario: foca en lo que hace y espera, no en detalles técnicos.

Cucumber JVM

- CucumberJVM es una herramienta para automatizar pruebas basadas en la metodología BDD (Behavior Driven Development), permitiendo escribir los escenarios en lenguaje natural (español o inglés) y vincularlos a código Java para validarlos automáticamente.

- ¿Cómo funciona CucumberJVM?
  - Los requisitos o funcionalidades esperadas se escriben en lenguaje Gherkin, usando archivos .feature.
  - Cada paso del escenario se vincula a un método Java concreto en las step definitions (definiciones de pasos).
  - Esto facilita la colaboración entre equipos técnicos y de negocio, ya que todos pueden leer y entender las pruebas.

Ejemplo Cucumber JVM

- Escenario en Gherkin (Suma.feature)

```
Feature: Suma de dos números
  Scenario: Usuario suma dos números
    Given que el usuario ingresa el número 4
    And ingresa el número 5
    When pulsa el botón sumar
    Then el resultado debe ser 9
```

- Definición de pasos en Java (StepDefinitions.java)

```java
import io.cucumber.java.es.Dado;
import io.cucumber.java.es.Cuando;
import io.cucumber.java.es.Entonces;
import static org.junit.Assert.*;

public class StepDefinitions {
    int numero1, numero2, resultado;

    @Dado("que el usuario ingresa el número {int}")
    public void que_el_usuario_ingresa_el_número(Integer numero) {
        if (numero1 == 0) {
            numero1 = numero;
        } else {
            numero2 = numero;
        }
    }

    @Cuando("pulsa el botón sumar")
    public void pulsa_el_boton_sumar() {
        resultado = numero1 + numero2;
    }

    @Entonces("el resultado debe ser {int}")
    public void el_resultado_debe_ser(Integer esperado) {
        assertEquals(esperado.intValue(), resultado);
    }
}
```

Webdriver

- El WebDriver es la pieza clave en Selenium para automatizar navegadores web desde código. Es una interfaz que permite controlar de forma programada un navegador, simulando acciones como abrir páginas, escribir texto, hacer clic, etc.

- ¿Qué es WebDriver?
  - Es una API que maneja distintas implementaciones específicas para cada navegador (ChromeDriver, GeckoDriver para Firefox, etc.).
  - Facilita la interacción con elementos de la página web usando selectores.
  - Funciona como puente entre tu código (Java, Python, etc.) y el navegador.

Conceptos clave de Webdriver

Concepto	Descripción
WebDriver	Interfaz para controlar el navegador
Driver específico	Programa que controla navegador concreto (p.ej. ChromeDriver)
Elemento Web	Cualquier componente HTML de la página
Selectores	Formas de encontrar elementos (por ID, nombre, clase, etiquetas...)

Pasos básicos para usar Webdriver

- Descargar Driver específico de tu navegador (p.ej. ChromeDriver para Chrome).
- Configurar la ruta al driver en tu proyecto.
- Crear una instancia de WebDriver para controlar el navegador.
- Navegar a una URL con .get("url").
- Buscar elementos de la página con métodos como .findElement(By.id("...")).
- Interactuar con esos elementos (escribir, hacer clic, leer texto).
- Cerrar el navegador con .quit() para liberar recursos.

Selenium

- ¿Qué es Selenium y para qué sirve?
  - Selenium es una herramienta de automatización de pruebas para aplicaciones web. Permite controlar navegadores (como Chrome, Firefox, Edge) mediante código, simulando acciones de un usuario real: abrir páginas, escribir textos, hacer clic, etc.
  - Automatiza pruebas repetitivas de interfaces web, asegurando que una web funciona correctamente tras cambios.
  - Es multiplataforma y compatible con muchos lenguajes (Python, Java, etc.).

Selenium

- Como agregar Selenium a tu proyecto?

```
<dependencies>
    <dependency>
        <groupId>org.seleniumhq.selenium</groupId>
        <artifactId>selenium-java</artifactId>
        <version>4.10.0</version>
    </dependency>
</dependencies>
```

- Descarga el WebDriver del navegador
- Por ejemplo, para Chrome, descarga el ChromeDriver y pon el ejecutable en una ruta accesible.

https://sites.google.com/chromium.org/driver/

Selenium

```java
public class SeleniumDemo {
    public static void main(String[] args) {
        // Configura la ruta al chromedriver
        System.setProperty("webdriver.chrome.driver", "C:/ruta/a/chromedriver.exe");

        // Crea instancia del navegador
        WebDriver driver = new ChromeDriver();

        // Abre Google
        driver.get("https://www.google.com");

        // Busca la caja de texto y escribe "Selenium"
        WebElement searchBox = driver.findElement(By.name("q"));
        searchBox.sendKeys("Selenium");
        searchBox.submit();

        // Espera unos segundos para ver resultados
        try {
            Thread.sleep(3000);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }

        // Muestra el título de la página
        System.out.println("Título actual: " + driver.getTitle());

        // Cierra el navegador
        driver.quit();
    }
}
```

- **System.setProperty** establece el path del driver del navegador.
- **new ChromeDriver()** abre una instancia de Chrome controlable.
- **driver.get(url)** abre una página web.
- **findElement(By.name("q"))** busca un elemento por su atributo name.
- **sendKeys y submit** simulan interacción del usuario.
- **Thread.sleep** espera para que cargue la página (para pruebas rápidas).
- **driver.getTitle()** obtiene el título de la ventana actual.
- **driver.quit()** cierra el navegador.

Testing Library

Tema 4

Testing Library

- Testing Library es una familia de bibliotecas enfocadas en testing para aplicaciones JavaScript, especialmente para interfaces de usuario.

- Su objetivo principal es probar componentes de forma que se asemeje a cómo los usuarios interactúan realmente con la aplicación, priorizando la accesibilidad y el comportamiento del usuario sobre la estructura interna.

Conceptos básicos

- **Enfoque en la experiencia del usuario**: Las pruebas no dependen de detalles internos como clases o estructuras complejas, sino que buscan elementos visibles (texto, roles accesibles).
- **Simular interacciones reales**: En lugar de invocar funciones internas, se simulan eventos reales como clics, tipeo, selección, etc.
- **Compatible con frameworks**: Hay versiones específicas para React, Vue, Angular, pero comparten la filosofía.

Funciones principales

- **render():** Renderiza el componente para pruebas.
- **screen:** Proporciona acceso a la pantalla virtual donde buscar elementos.
- **getByText(), getByRole(), getByLabelText():** Métodos para localizar elementos visibles y accesibles.
- **fireEvent y userEvent:** Para simular eventos de usuario (click, cambio, teclado).
- Uso de **await** con funciones asíncronas para esperar resultados que dependen de procesos.

Ejemplo con React Testing Library

```
import { render, screen } from '@testing-library/react';
import userEvent from '@testing-library/user-event';
import MiComponente from './MiComponente';

test('el botón cambia texto al hacer click', async () => {
  render(<MiComponente />);
  const boton = screen.getByRole('button', { name: /click aquí/i });

  await userEvent.click(boton);

  expect(screen.getByText(/texto cambiado/i)).toBeInTheDocument();
});
```

Pruebas de acesibilidad

Tema 5

Pruebas de accesibilidad

- En el testing front end, las pruebas de accesibilidad son fundamentales para garantizar que las aplicaciones web sean utilizables por todas las personas, incluyendo usuarios con discapacidades visuales, auditivas, motoras o cognitivas.

- Estas pruebas se enfocan en verificar que el contenido y la interfaz cumplan con estándares y buenas prácticas para que herramientas como lectores de pantalla, navegación con teclado y otros dispositivos de asistencia funcionen correctamente.

Aspectos clave en pruebas de accesibilidad

- Comprobar que los elementos tengan roles accesibles y etiquetas ARIA correctamente usados.
- Validar el orden lógico de tabulación para navegación con teclado.
- Verificar suficiente contraste de colores y tamaño legible de fuentes.
- Asegurar que todo contenido visual tenga equivalentes en texto o audio.
- Evaluar el uso correcto de landmarks y estructuras semánticas.

Herramientas y librerías comunes

- **Testing Library + jest-dom + jest-axe**: Para pruebas automáticas en código con integración directa en flujos de testing.
- **axe-core**: Biblioteca robusta que detecta automáticamente problemas de accesibilidad.
- **WAVE**: Extensión visual para navegadores que destaca errores accesibles.
- **Lighthouse**: Auditoría integrada en Chrome para verificar accesibilidad y rendimiento.

Ejemplo con Testing Library y jest-axe

```
import { render } from '@testing-library/react';
import { axe, toHaveNoViolations } from 'jest-axe';
import MiComponente from './MiComponente';

expect.extend(toHaveNoViolations);

test('MiComponente debería ser accesible', async () => {
  const { container } = render(<MiComponente />);
  const resultados = await axe(container);
  expect(resultados).toHaveNoViolations();
});
```

Integración DevOps

Tema 6

Integración Devops

- La integración de pruebas automatizadas dentro de un pipeline DevOps es fundamental para garantizar calidad y agilidad en la entrega de software. Se trata de incorporar la ejecución continua y automática de pruebas (unitarias, de integración, de aceptación, etc.) en las etapas de construcción, despliegue y liberación del software.

Conceptos clave

- **Automatización continua**: Cada cambio en el código dispara la ejecución de pruebas para detectar errores tempranamente.
- **Feedback rápido**: Los desarrolladores reciben resultados rápidos para corregir fallas antes de avanzar.
- **Ciclo de vida completo**: Las pruebas se ejecutan en todos los entornos (desarrollo, staging, producción).
- **Cobertura amplia**: Se combinan pruebas unitarias, integración, end to end y de aceptación para mayor confianza.
- **Orquestación**: Herramientas como Jenkins, GitHub Actions, GitLab CI/CD, Azure DevOps, Bamboo, entre otros, coordinan estas tareas.

Pasos comunes para integrar pruebas en DevOps

1. Configurar repositorio con pruebas automatizadas (Jest, Mocha, Selenium, Cypress).
2. Definir pipeline CI/CD donde se instalen dependencias, se ejecuten pruebas automáticamente tras cada commit o pull request.
3. Incluir reportes de pruebas y métricas para análisis en dashboards.
4. Establecer reglas para bloquear merges o despliegues si las pruebas fallan.
5. Implementar pruebas en entornos reales o simulados mediante contenedores (Docker) y entornos efímeros.

Beneficios

- Mayor calidad y velocidad en lanzamientos.
- Detección temprana de fallos.
- Mejora de la colaboración entre desarrollo y operaciones.

Jenkins

- Jenkins es una herramienta open source escrita en Java que permite automatizar tareas repetitivas dentro del desarrollo de software, especialmente enfocada en la Integración Continua (CI) y Entrega Continua (CD). Facilita la construcción, prueba y despliegue automático de proyectos, detectando errores tempranamente y mejorando la calidad y velocidad de entrega.

- ¿Qué es Jenkins?
  - Un servidor de automatización para CI/CD.
  - Utiliza una arquitectura de servidor y nodos para distribuir tareas.
  - Permite definir trabajos (jobs) y pipelines para automatizar flujos completos.
  - Extensible con miles de plugins que integran Git, Docker, Kubernetes y más.

Jenkins

- **Instalación en Windows**
  - Descarga el instalador oficial de Jenkins (.msi) desde jenkins.io.
  - Ejecuta el instalador y sigue el asistente (asegúrate de tener Java instalado, Jenkins requiere Java).
  - El instalador inicia Jenkins como un servicio de Windows.
  - Accede a Jenkins abriendo el navegador en http://localhost:8080.
  - Copia la contraseña inicial que aparecerá en el archivo indicado para desbloquear Jenkins.
  - Sigue el asistente web para instalar plugins recomendados y crear el primer usuario.

- Si tienes Docker instalado, puedes ejecutar Jenkins en un contenedor:

```
docker run -d -p 8080:8080 -p 50000:50000 --name jenkins jenkins/jenkins:lts
```

Integrar Jenkins con Git

Paso	Acción
Instalar plugin Git	Administrar Jenkins > Plugins > Git plugin
Crear job	Nuevo Item > Proyecto estilo libre
Configurar Git	Gestión código fuente > Git URL + credenciales
Configurar disparadores	Poll SCM o webhooks para ejecuciones automáticas
Definir pasos de construcción	Scripts o comandos para build/test
Configurar webhook en GitHub	Para notificaciones inmediatas

Integrar Jenkins con Maven

- Prerrequisitos:
  - Jenkins instalado y corriendo.
  - Plugin Git instalado en Jenkins (como en pasos anteriores).
  - Maven instalado en la máquina donde corre Jenkins o configurado en Jenkins.
  - Tener un proyecto Java con archivo pom.xml en un repositorio Git.

Paso	Acción
Configurar Maven Jenkins	En Configuración global de herramientas
Crear Job	Nuevo Item > Proyecto estilo libre
Añadir Git	URL repositorio + credenciales
Añadir construcción Maven	Paso construir con objetivo `clean install`
Configurar disparadores	Poll SCM o webhook Git
Ejecutar build	Construir ahora y revisar logs

Integrar Jenkins con pruebas

- Para integrar Jenkins con pruebas automáticas en un proyecto Maven usando Git, debes configurar que Jenkins ejecute las pruebas como parte del proceso de construcción. Maven ejecuta pruebas automáticamente con el comando mvn test o dentro del ciclo mvn clean install

Paso	Acción
Ejecutar pruebas con Maven	Usar `clean test` o `clean install`
Publicar resultados en Jenkins	Post-construcción > Publicar resultados JUnit
Configurar path reportes	`**/target/surefire-reports/*.xml`
Visualizar en Jenkins	Métricas y detalles de pruebas pasadas/ fallidas

Pipelines en Jenkins

- Los pipelines en Jenkins son un conjunto de pasos o etapas definidas en código que permiten automatizar todo el ciclo de vida de la construcción, prueba y despliegue de software. Son flujos de trabajo codificados que facilitan la creación, visualización y control de procesos complejos de integración y entrega continua (CI/CD).

- Qué son pipelines en Jenkins
  - Definen procesos automatizados complejos, como: compilar código, ejecutar pruebas, análisis de calidad, despliegues, notificaciones.
  - Están codificados usualmente en un archivo llamado Jenkinsfile, que puede estar versionado junto con el código en Git.
  - Permiten mayor control, modularidad, y reproducibilidad en el proceso de DevOps.
  - Pueden ser declarativos (más estructurados) o scripted (más flexibles).

Crear Pipelines en Jenkins

Paso	Descripción
Configurar Maven	En herramientas globales de Jenkins
Preparar Jenkinsfile	Definir etapas checkout y build con Maven
Crear pipeline	Nuevo item > Pipeline > pipeline script from SCM
Ejecutar pipeline	Construir ahora y revisar ejecución

Sonar

- Integrar SonarQube con Jenkins es una práctica común para analizar la calidad del código automáticamente en pipelines. Aquí tienes una guía básica para usar SonarQube junto con Jenkins y Maven.

- Qué es SonarQube?
  - Plataforma para análisis estático que detecta bugs, vulnerabilidades y code smells.
  - Genera reportes completos de calidad del código.
  - Se integra con muchas herramientas CI/CD, incluyendo Jenkins.

Instalar SonarQube

- Requisitos previos
  - Java 11 o superior instalado en tu equipo (SonarQube lo necesita para funcionar).
  - Al menos 2 GB de memoria RAM recomendada.

- **Paso 1: Descargar SonarQube**
  - Ve a la página oficial de descargas: https://www.sonarqube.org/downloads/
  - Descarga la versión Community (gratuita) o la que necesites.
  - Descomprime el archivo descargado en la ruta donde quieras instalar SonarQube

Instalar SonarQube

- **Paso 2: Configurar SonarQube**

1. Entra a la carpeta conf dentro de la instalación:

   cd /opt/sonarqube/conf

2. Edita sonar.properties para configurar opciones como el puerto o la base de datos (por defecto usa una base de datos embebida para pruebas).

   Ejemplo para cambiar el puerto a 9000 si es necesario:

   sonar.web.port=9000

Instalar SonarQube

- **Paso 3: Iniciar SonarQube**

1. Ve a la carpeta /bin y elige el script según tu SO, por ejemplo para Linux 64-bit:

```
cd /opt/sonarqube/bin/linux-x86-64/
```

2. Ejecuta el siguiente comando para iniciar SonarQube:

```
./sonar.sh start
```

3. Verifica el estado con:

```
./sonar.sh status
```

Instalar SonarQube

- **Paso 4: Acceder a SonarQube**
  - Abre tu navegador y entra a: http://localhost:9000
  - Usuario y contraseña por defecto son ambos: admin
  - Cambia la contraseña en el primer inicio para mayor seguridad.

- **Paso 5: (Opcional) Configurar base de datos externa**
  - Para producción, mejor usar bases de datos como PostgreSQL, en sonar.properties puedes configurar:

```
sonar.jdbc.username=usuario
sonar.jdbc.password=contraseña
sonar.jdbc.url=jdbc:postgresql://localhost/sonarqube
```

Icono *Training Consulting*

Testing Front End

Completa nuestra encuesta de satisfacción a través del QR

GRACIAS

2025