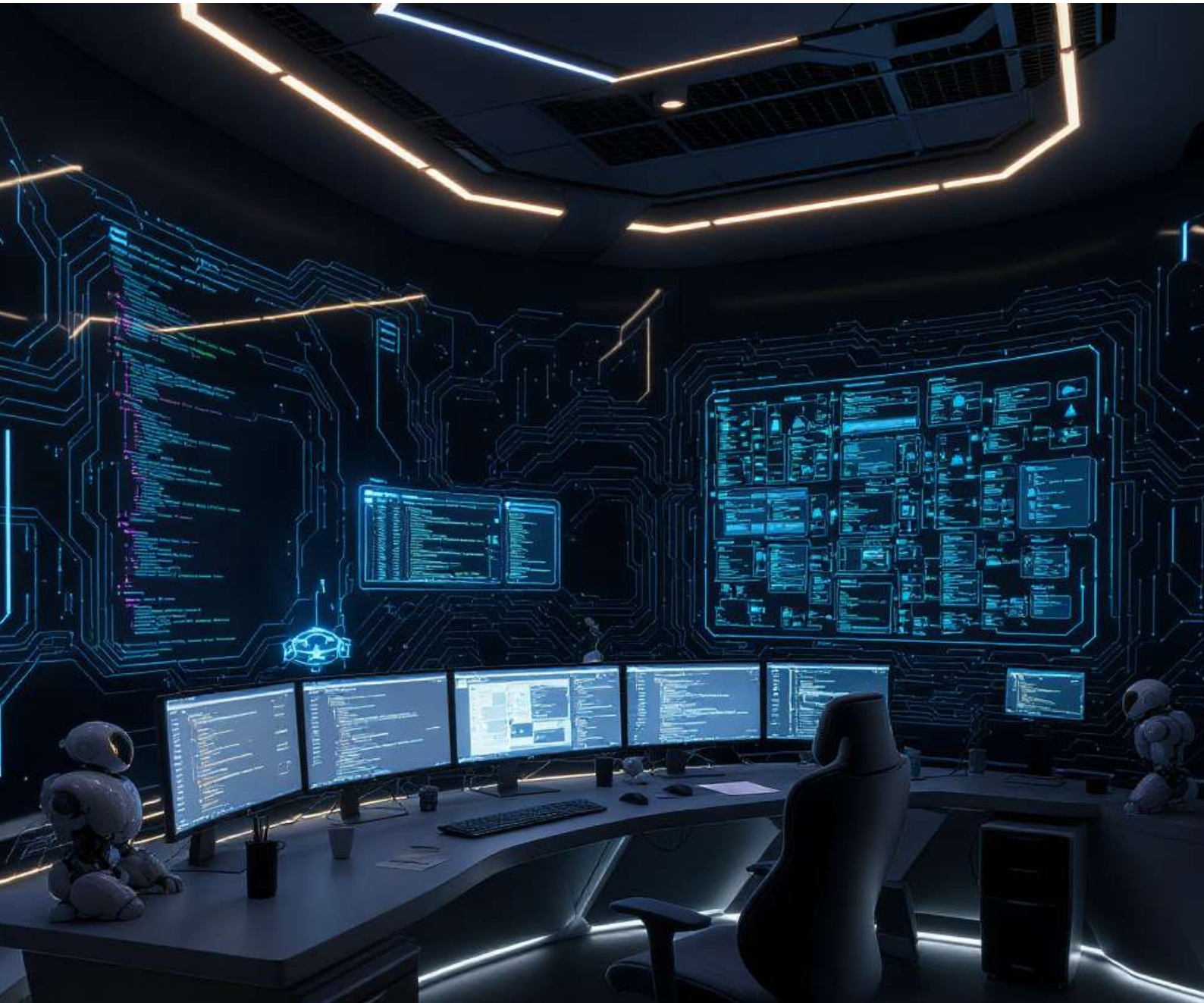
{desafío}
latam_

CURSO

TEST AUTOMATION ENGINEER
FORMACIÓN INTEGRAL



Objetivo General del Curso

ANALIZAR LAS HERRAMIENTAS DE DISEÑO DE TEST AUTOMATION ENGINEER, DE ACUERDO A LAS APLICACIONES WEB, MÓVILES Y APIS.

Objetivo específico del Módulo

UTILIZAR HERRAMIENTAS DE DOCKER, ENTORNOS VIRTUALIZADOS Y PRUEBAS EN LA NUBE, DE ACUERDO A LAS APLICACIONES WEB, MÓVILES Y APIS.

Contenidos

Módulo 11: HERRAMIENTAS DE DOCKER, ENTORNOS VIRTUALIZADOS Y PRUEBAS EN LA NUBE

Capítulo 1: FUNDAMENTOS DE CONTENEDORES

Los contenedores son una de las tecnologías clave en el desarrollo y despliegue moderno de software. Permiten encapsular aplicaciones y sus dependencias en una unidad ejecutable que puede ejecutarse de manera coherente en diferentes entornos. Esta capacidad de portabilidad, aislamiento y eficiencia ha transformado los procesos de integración, pruebas y entrega continua en proyectos de software profesional.

¿Qué es un contenedor?

Un contenedor es un entorno de ejecución ligero, portátil y autosuficiente que encapsula una aplicación junto con todas sus dependencias: librerías, archivos de configuración, herramientas del sistema, etc.

Los contenedores comparten el kernel del sistema operativo del host, a diferencia de las máquinas virtuales, que requieren su propio sistema operativo completo.

Diferencias entre contenedor y máquina virtual

CARACTERÍSTICA	CONTENEDOR	MÁQUINA VIRTUAL
Aislamiento	A nivel de proceso y sistema de archivos	A nivel de sistema operativo completo
Arranque	Rápido (segundos)	Lento (minutos)
Consumo de recursos	Bajo	Alto
Portabilidad	Alta (ejecutable en cualquier host con motor de contenedores)	Media (requiere hipervisor compatible)
Caso de uso común	Microservicios, pruebas, CI/CD	Simulación de entornos completos

¿Por qué son útiles los contenedores en QA y testing?

En entornos de pruebas automatizadas y QA, los contenedores permiten:

- Reproducibilidad: el mismo contenedor puede ejecutarse en cualquier entorno (local, CI, nube).
- Aislamiento: cada suite de pruebas puede ejecutarse en su propio entorno sin interferencias.
- Eficiencia: se pueden levantar y destruir entornos de prueba en segundos.
- Escalabilidad: múltiples contenedores pueden ejecutarse en paralelo para pruebas concurrentes.

Facilidad de integración en CI/CD: herramientas como Jenkins, GitHub Actions o GitLab CI soportan nativamente la ejecución de pruebas en contenedores.

Componentes clave de los contenedores

Imagen

Una imagen es una plantilla inmutable que contiene el sistema de archivos y las instrucciones necesarias para crear un contenedor. Es el "plano" del contenedor.

Contenedor

Una instancia en ejecución de una imagen. Cada vez que se ejecuta una imagen, se crea un contenedor.

Dockerfile

Archivo de texto que contiene las instrucciones para construir una imagen. Define el entorno y los pasos para preparar la aplicación.

Ejemplo básico:

```
dockerfile

FROM node:18
WORKDIR /app
COPY . .
RUN npm install
CMD ["npm", "run", "test"]
```

Volumen

Un volumen permite persistir datos fuera del ciclo de vida del contenedor (para logs, resultados de pruebas, configuraciones).

Red

Los contenedores pueden comunicarse entre sí mediante redes virtuales internas, lo que permite simular arquitecturas reales (cliente-servidor, microservicios).

Flujo de trabajo básico en testing con contenedores

- Escribir un Dockerfile con la configuración del entorno de pruebas.
- Construir la imagen con docker build.
- Ejecutar el contenedor con docker run.
- Ejecutar pruebas dentro del contenedor.
- Obtener resultados y reportes.
- Eliminar contenedor al finalizar para liberar recursos.

Integración con herramientas de automatización

Los contenedores se integran naturalmente con herramientas como:

- Cypress: ejecución de pruebas E2E dentro de contenedores headless.
- Postman + Newman: ejecución de pruebas API automatizadas en entornos CI/CD.
- Playwright, Puppeteer: testing visual y funcional.
- Jest + Supertest: pruebas unitarias e integración.
- Docker Compose: para levantar múltiples contenedores (por ejemplo, app + base de datos).

Ventajas en entornos en la nube

En plataformas como AWS, Azure, Google Cloud o servicios especializados (BrowserStack, Sauce Labs), los contenedores permiten:

- Ejecutar pruebas en paralelo bajo demanda.
- Escalar automáticamente según la carga de pruebas.
- Disponer de entornos temporales con configuraciones específicas.
- Integrar imágenes propias desde registros como DockerHub o GitHub Container Registry.

Los contenedores son una base sólida para implementar flujos de pruebas confiables, reproducibles y eficientes. Su uso en combinación con pipelines CI/CD, herramientas de testing automatizado y entornos en la nube constituye una práctica moderna y profesional en el aseguramiento de calidad del software.

Dominar los fundamentos de contenedores es esencial para cualquier QA o ingeniero en automatización.

Capítulo 2: CONSTRUCCIÓN Y EJECUCIÓN DE IMÁGENES CON DOCKER

Las imágenes Docker son el corazón de la tecnología de contenedores. Representan un entorno inmutable y portable que contiene todo lo necesario para ejecutar una aplicación o suite de pruebas: sistema base, dependencias, código, configuraciones y scripts. Comprender cómo construir y ejecutar imágenes es esencial para automatizar pruebas en entornos consistentes, replicables y controlados.

¿Qué es una imagen Docker?

Una imagen Docker es un archivo binario compuesto por una serie de capas que definen un sistema de archivos completo. Es una plantilla que se puede ejecutar como un contenedor.

Cada vez que se modifica una capa (por ejemplo, al instalar una dependencia), se genera una nueva versión con una nueva capa sobre la anterior. Esto permite reutilización y almacenamiento eficiente.

¿Qué es un Dockerfile?

Un Dockerfile es un archivo de texto que contiene instrucciones para construir una imagen Docker personalizada. Cada instrucción representa una nueva capa en la imagen final.

Estructura básica de un Dockerfile

Ejemplo: imagen para pruebas de Node.js con Jest

```dockerfile
# Imagen base
FROM node:18

# Directorio de trabajo dentro del contenedor
WORKDIR /app

# Copiar archivos del proyecto al contenedor
COPY . .

# Instalar dependencias
RUN npm install

# Comando por defecto al ejecutar el contenedor
CMD ["npm", "test"]
```

Construcción de una imagen

Una vez creado el Dockerfile, la imagen se construye usando el comando:

```bash
docker build -t nombre-de-la-imagen .
```

- -t: etiqueta (tag) de la imagen.
- .: indica que el contexto de construcción está en el directorio actual.

Ejemplo:

```bash
docker build -t qa-suite:v1 .
```

Ejecución de una imagen como contenedor

Una vez construida la imagen, se ejecuta como un contenedor usando:

```bash
docker run [opciones] nombre-de-la-imagen
```

Ejemplo básico:

```bash
docker run --rm qa-suite:v1
```

- --rm: elimina el contenedor después de la ejecución.

El contenedor correrá el comando definido en CMD (en este caso, npm test).

Comandos útiles durante la construcción y ejecución

Comando	Descripción
docker images	Lista las imágenes locales disponibles.
docker ps -a	Lista contenedores activos y finalizados.
docker logs <container_id>	Muestra la salida del contenedor.
docker exec -it <container_id> bash	Accede de forma interactiva al contenedor.
docker rmi <imagen>	Elimina una imagen.
docker build --no-cache -t imagen .	Fuerza reconstrucción ignorando la caché.

Mejores prácticas para construcción de imágenes

- Utilizar imágenes base ligeras como node:alpine, python:slim, etc.
- Minimizar capas (combinar RUN cuando sea posible).
- Excluir archivos innecesarios usando .dockerignore.
- Evitar almacenar credenciales en la imagen.
- Fijar versiones de dependencias para evitar cambios inesperados.

Aplicaciones en QA y automatización

Las imágenes Docker permiten:

- Construir contenedores que ejecutan suites de pruebas (Jest, Cypress, Newman, etc.).
- Integrarse con pipelines CI/CD para ejecutar pruebas automáticamente al hacer push.
- Simular entornos controlados de staging.
- Ejecutar pruebas en paralelo con múltiples contenedores.
- Compartir entornos de prueba estandarizados entre equipos.

Ejemplo práctico: Cypress en Docker

dockerfile

```
FROM cypress/included:12.17.1
WORKDIR /e2e
COPY . .
CMD ["npx", "cypress", "run"]
```

Build & Run:

bash

```
docker build -t cypress-tests .
docker run --rm cypress-tests
```

Esto ejecuta una batería de pruebas E2E de Cypress sin depender de configuraciones locales del sistema operativo.

Construir y ejecutar imágenes con Docker permite encapsular entornos de pruebas de forma eficiente, portable y controlada. Esta práctica es clave para escalar pruebas en pipelines CI/CD, garantizar entornos coherentes y automatizar validaciones en múltiples etapas del desarrollo moderno. El dominio de esta técnica es esencial para cualquier profesional que trabaje con automatización de calidad de software.

Capítulo 3: USO DE CONTENEDORES PARA EJECUTAR PRUEBAS

El uso de contenedores para ejecutar pruebas automatizadas ha transformado las estrategias de aseguramiento de calidad. Gracias a su portabilidad, aislamiento y rapidez de despliegue, los contenedores permiten simular entornos productivos de forma controlada, estandarizada y reproducible, lo que resulta ideal para pipelines CI/CD, pruebas paralelas y entornos de staging.

¿Por qué usar contenedores para ejecutar pruebas?

Ventajas principales:

- Aislamiento de entornos: cada prueba se ejecuta en su propio entorno, libre de interferencias externas.
- Reproducibilidad: los tests se comportan igual en cualquier entorno que ejecute el contenedor.
- Velocidad: los contenedores arrancan en segundos.
- Escalabilidad: múltiples suites de prueba pueden ejecutarse en paralelo.
- Facilidad de integración con CI/CD: herramientas como GitHub Actions, GitLab CI y Jenkins ejecutan pruebas en contenedores nativamente.

Tipos de pruebas que se benefician del uso de contenedores

Tipo de prueba	Beneficio con contenedores
Unitarias	Aislamiento completo del entorno de ejecución.
Integración	Facilita la conexión con servicios como bases de datos o APIs.
End-to-End (E2E)	Permite simular ambientes reales con múltiples servicios.
API Testing	Se puede mockear, interceptar o validar servicios REST.
Pruebas en paralelo	Ideal para ejecutar múltiples suites simultáneamente.

Flujo básico: ejecutar pruebas dentro de un contenedor

- Definir un Dockerfile que incluya las dependencias necesarias para la ejecución de pruebas.
- Construir la imagen con el entorno de testing.
- Ejecutar el contenedor con los comandos de prueba como punto de entrada.
- Capturar resultados o reportes de salida desde volúmenes compartidos o archivos.
- Destruir el contenedor al finalizar.

Ejemplo práctico: pruebas E2E con Cypress en Docker

Estructura:

```dockerfile
FROM cypress/included:12.17.1
WORKDIR /e2e
COPY . .
CMD ["npx", "cypress", "run"]
```

Construcción y ejecución:

```bash
docker build -t cypress-suite .
docker run --rm -v $PWD/results:/e2e/cypress/reports cypress-suite
```

Esto:

- Ejecuta la suite de pruebas.
- Guarda los resultados en la carpeta results/ de la máquina host.
- Se autodestruye al finalizar.

Ejemplo práctico: pruebas de API con Postman + Newman

Dockerfile:

```dockerfile
FROM postman/newman:alpine
COPY collection.json /etc/postman/
CMD ["run", "/etc/postman/collection.json"]
```

Build & Run:

```bash
docker build -t api-tests .
docker run --rm api-tests
```

Opcionalmente, se pueden añadir entornos (-e) y reportes (--reporters html).

Buenas prácticas en pruebas con contenedores

- Separar datos de pruebas mediante volúmenes o fixtures montados.
- Definir scripts de entrada (CMD, ENTRYPOINT) claros y orientados a pruebas.
- Exportar logs o reportes al sistema host o a sistemas de monitoreo.
- Usar variables de entorno para parametrizar comportamientos.
- Mantener las imágenes livianas para reducir tiempos de construcción y ejecución.
- Utilizar etiquetas (tags) por versión de pruebas para trazabilidad.

Casos de uso reales

- Ejecutar pruebas automatizadas cada vez que se realiza un push en GitHub.
- Validar microservicios que se despliegan como contenedores junto a sus dependencias.
- Correr pruebas en paralelo en pipelines distribuidos (Docker + Selenium Grid, Cypress Dashboard).
- Simular flujos con múltiples contenedores usando Docker Compose.

Integración con CI/CD

Plataformas como GitHub Actions o GitLab CI permiten ejecutar pruebas en contenedores como parte de sus jobs. Ejemplo de paso en un workflow:

```yaml
- name: Ejecutar pruebas en contenedor
  run: docker run --rm -v ${{ github.workspace }}/resultados:/app/results mi-imagen-test
```

Ejecutar pruebas dentro de contenedores permite estandarizar, escalar y automatizar los procesos de testing de forma confiable. Es una estrategia fundamental para DevOps, QA moderno y cualquier flujo de desarrollo continuo. Su adopción mejora la trazabilidad, estabilidad y reproducibilidad de las pruebas en todos los entornos.

Capítulo 4: EXPLORACIÓN DE BROWSERSTACK, SAUCE LABS Y AWS DEVICE FARM

En un entorno de desarrollo ágil, donde la diversidad de dispositivos, navegadores y sistemas operativos es cada vez mayor, las herramientas de testing en la nube se han convertido en un estándar para asegurar compatibilidad y funcionalidad en condiciones reales. Plataformas como BrowserStack, Sauce Labs y AWS Device Farm permiten ejecutar pruebas automatizadas o manuales en dispositivos físicos o entornos virtuales sin necesidad de infraestructura propia.

Estas herramientas ofrecen soporte para frameworks como Selenium, Appium, Playwright, Cypress, Espresso, XCTest, entre otros.

BrowserStack

BrowserStack es una plataforma de testing en la nube que permite ejecutar pruebas web y móviles en más de 3.000 navegadores y dispositivos reales. Es ampliamente utilizada en QA para realizar pruebas de regresión, compatibilidad cruzada y validación de interfaces responsivas.

Características clave:

- Soporte para Selenium, Appium, Cypress, Playwright.
- Testing manual o automatizado.
- Dispositivos reales (iOS, Android) y navegadores (Chrome, Safari, Firefox, Edge).
- Capturas de pantalla, grabaciones y logs automáticos.
- Integración con herramientas CI/CD: GitHub Actions, Jenkins, CircleCI.

Ejemplo de ejecución (Selenium):

```python
desired_cap = {
  'browser': 'Chrome',
  'browser_version': 'latest',
  'os': 'Windows',
  'os_version': '10',
  'name': 'Ejemplo Selenium en BrowserStack'
}

driver = webdriver.Remote(
  command_executor='https://usuario:clave@hub-cloud.browserstack.com/wd/hub',
  desired_capabilities=desired_cap
)
```

Sauce Labs

Sauce Labs ofrece una plataforma similar a BrowserStack, con foco tanto en pruebas funcionales como de rendimiento, accesibilidad y rendimiento visual. Ofrece integración con laboratorios de dispositivos reales y simuladores.

Características clave:

- Más de 800 combinaciones de navegador/SO.
- Dispositivos físicos para pruebas móviles reales.
- Integración con Selenium, Appium, Cypress y TestCafe.
- Herramientas de performance (Sauce Performance) y calidad visual (Sauce Visual).
- Dashboards de análisis de fallos, métricas e historial de builds.

Ejemplo de integración en CI:

```yaml
- name: Run tests on Sauce Labs
  run: |
    saucectl run --config .sauce/config.yml
```

AWS Device Farm

AWS Device Farm es un servicio de Amazon Web Services que permite ejecutar pruebas automatizadas en una gran variedad de dispositivos físicos Android e iOS hospedados en la nube de AWS. Ideal para pruebas móviles más exigentes en entornos empresariales.

Características clave:

- Dispositivos reales, no emuladores.
- Soporte para Appium, Calabash, UIAutomator, Espresso, XCTest.
- Subida de APK/IPA o ejecución desde scripts automatizados.
- Análisis automático de resultados (logs, videos, screenshots, métricas de rendimiento).
- Integración con pipelines de CI/CD vía AWS CLI o APIs.

Flujo básico:

- Subir la app (APK o IPA).
- Seleccionar dispositivos.
- Configurar tipo de test (Appium, manual, exploratorio).
- Descargar reportes con métricas detalladas.

Comparación general

Característica	BrowserStack	Sauce Labs	AWS Device Farm
Dispositivos móviles	Reales y emulados	Reales y emulados	Solo reales
Navegadores web	Sí	Sí	No
Soporte para Appium	Sí	Sí	Sí
Soporte para Cypress	Sí	Sí	Parcial (indirecto)
CI/CD integrations	Alta (Jenkins, GitHub, etc.)	Alta	Alta (AWS-focused)
Escalabilidad empresarial	Media/Alta	Alta	Muy alta (infraestructura AWS)

Ventajas del testing en la nube con estas herramientas

- Acceso inmediato a una gran variedad de dispositivos físicos sin costos de infraestructura.
- Pruebas distribuidas en paralelo, acelerando la validación de versiones.
- Recolección automática de evidencia: logs, screenshots, videos.
- Integración directa en pipelines CI/CD para pruebas continuas.
- Cobertura real del usuario final, mejorando la experiencia y reduciendo errores en producción.

El uso de plataformas como BrowserStack, Sauce Labs y AWS Device Farm permite a los equipos de QA y desarrollo validar sus aplicaciones en entornos reales y altamente controlados, sin las barreras físicas o de infraestructura que implican los laboratorios locales. Son herramientas fundamentales para lograr una estrategia de testing robusta, escalable y compatible con DevOps.

Capítulo 5: GESTIÓN DE ENTORNOS AISLADOS PARA QA

En el aseguramiento de calidad moderno, uno de los desafíos recurrentes es garantizar la estabilidad, reproducibilidad y control del entorno de pruebas. Para enfrentar esta problemática, se recurre a la gestión de entornos aislados, donde cada ciclo de pruebas se ejecuta en un espacio controlado, separado del resto del sistema o aplicaciones.

El aislamiento permite minimizar interferencias externas, asegurar la coherencia entre ejecuciones y simular condiciones reales de producción, lo que resulta esencial en QA técnico, testing automatizado y despliegues CI/CD.

¿Qué es un entorno aislado en QA?

Un entorno aislado es una instancia independiente de ejecución (física o virtual) que contiene:

- El sistema operativo base.
- Las versiones específicas del software bajo prueba.
- Las herramientas de testing.
- Las configuraciones, datos de prueba y servicios necesarios.

El entorno no comparte recursos críticos (como configuraciones globales, puertos, o dependencias) con otros sistemas o pruebas en curso.

Beneficios del aislamiento en entornos de pruebas

- Reproducibilidad: cada ejecución parte del mismo estado inicial.
- Prevención de conflictos: evita interferencias entre pruebas concurrentes.
- Seguridad: separa pruebas con datos sensibles.
- Escalabilidad: permite ejecutar múltiples suites en paralelo.
- Trazabilidad: los fallos pueden vincularse a una configuración o entorno específico.

Principales estrategias de aislamiento

1.- Contenedores (Docker)

- Encapsulan aplicaciones y dependencias en una unidad portable.
- Permiten ejecutar múltiples entornos en paralelo sin conflictos.
- Se integran fácilmente en pipelines y plataformas CI/CD.

Ejemplo:

```bash
docker run --rm -v $PWD/tests:/app cypress/included:12.17.1
```

2.- Máquinas virtuales

- Simulan un sistema completo, ideal para pruebas con dependencia de hardware, SO o drivers.
- Requieren más recursos que los contenedores.
- Útiles para testing multiplataforma (Windows, Linux, macOS).

3.- Entornos cloud (SaaS)

- BrowserStack, Sauce Labs, AWS Device Farm.
- Permiten probar en dispositivos físicos reales sin infraestructura propia.
- Ideal para testing cruzado y validación móvil.

Herramientas para gestionar entornos aislados en QA

Herramienta	Tipo de aislamiento	Casos de uso principales
Docker	Contenedor	Pruebas automatizadas, CI/CD
Docker Compose	Multicontenedor	Testing de microservicios
Vagrant + VirtualBox	Máquina virtual	Simulación completa de entornos
Kubernetes	Orquestación de pods	Testing distribuido y escalable
BrowserStack / Sauce Labs	Cloud (dispositivos reales)	Validación multiplataforma y móvil

Buenas prácticas para entornos aislados en QA

- **Inmutabilidad**: los entornos deben reconstruirse desde cero si hay fallos o actualizaciones.
- **Versionamiento de entornos**: almacenar versiones exactas de imágenes, scripts, dependencias.
- **Uso de volúmenes temporales**: evitar persistencia de resultados dentro del contenedor.
- **Separación de datos sensibles**: utilizar variables de entorno o secretos encriptados.
- **Destrucción automática**: asegurar la limpieza tras finalizar la ejecución (e.g., --rm en Docker).

Caso de uso: entorno aislado para pruebas E2E de un frontend React

- Se define un contenedor con Node.js, Cypress y dependencias de la app.
- Se copia el código fuente y los casos de prueba al contenedor.
- Se ejecuta npx cypress run.
- Los resultados se exportan a una carpeta externa mediante un volumen.
- El contenedor se elimina automáticamente al finalizar.

Este enfoque permite que las pruebas se ejecuten en cualquier máquina, con resultados consistentes, sin alterar el sistema operativo anfitrión.

Integración en pipelines CI/CD

Los entornos aislados son fundamentales para ejecutar pruebas automáticas como parte del ciclo de integración y despliegue continuo:

- Se crea una imagen con el entorno de pruebas.
- Se ejecuta en cada push, merge o deploy.
- Se destruye al finalizar, dejando trazabilidad de logs y reportes.

Ejemplo (GitHub Actions):

```yaml
- name: Ejecutar pruebas aisladas
  run: docker run --rm mi-imagen-pruebas
```

La gestión de entornos aislados es una práctica indispensable para garantizar la calidad, estabilidad y seguridad de los procesos de testing modernos. Ya sea a través de contenedores, máquinas virtuales o servicios cloud, su implementación permite mejorar la reproducibilidad de las pruebas, reducir los errores por interferencias y acelerar el feedback en el desarrollo ágil.