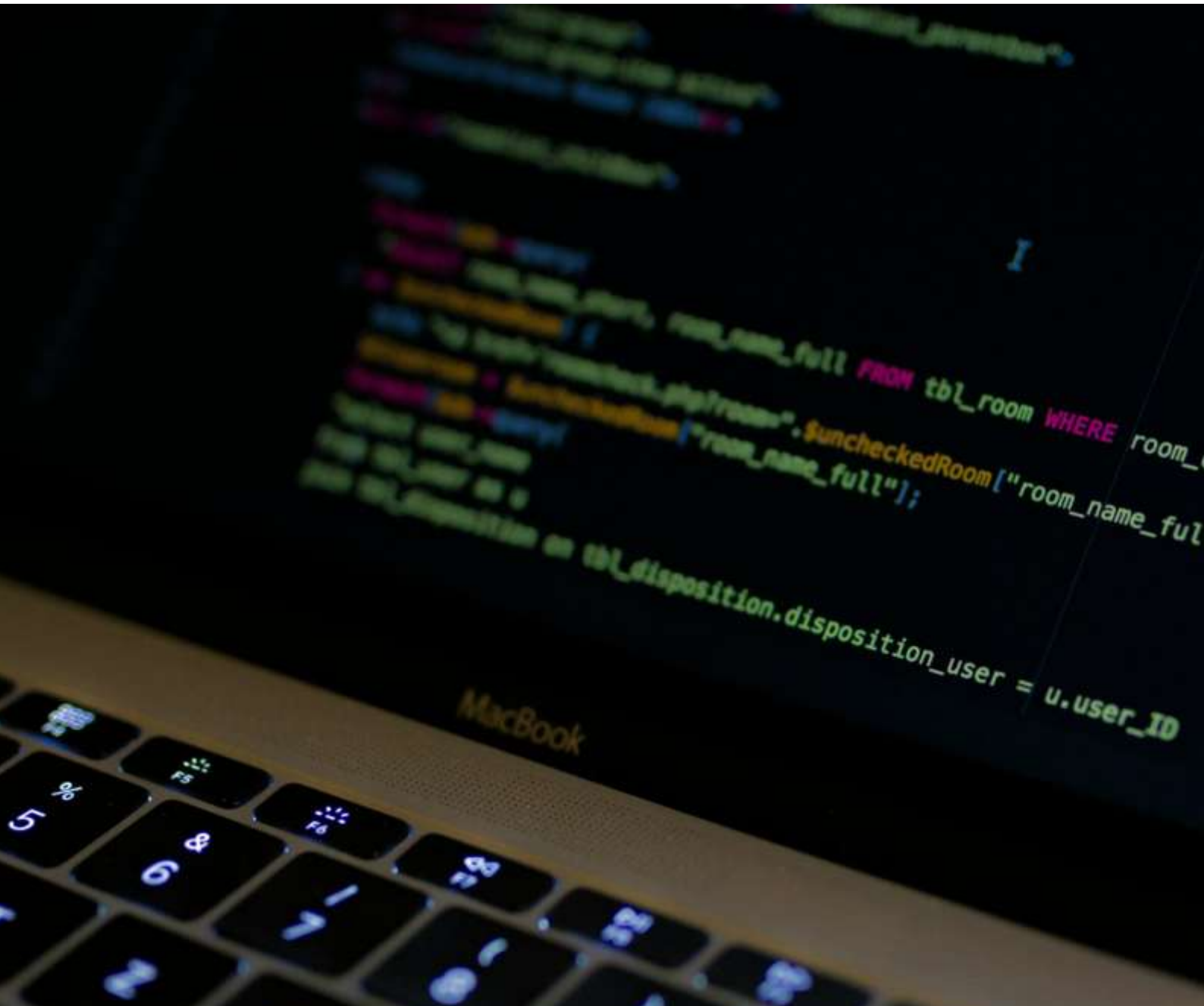
{desafío} latam_

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 AUTOMATIZACIÓN MÓVIL CON APPIUM, DE ACUERDO A LAS APLICACIONES WEB, MÓVILES Y APIS.

# Contenidos

# Módulo 9, HERRAMIENTAS DE AUTOMATIZACIÓN MÓVIL CON APPIUM

# Capítulo 1: Testing en entornos móviles: ANDROID y REACT NATIVE

El testing en entornos móviles presenta desafíos únicos respecto al testing tradicional en entornos web o de escritorio. Las aplicaciones móviles deben funcionar correctamente en una gran variedad de dispositivos con diferentes resoluciones, capacidades de hardware, versiones del sistema operativo y condiciones de red. Esta diversidad exige enfoques y herramientas de prueba robustas y especializadas.

## Características y desafíos del testing móvil

### Fragmentación del entorno

- **Dispositivos:** gran cantidad de marcas y modelos con diferentes tamaños de pantalla, resoluciones y prestaciones.
- **Sistemas operativos:** múltiples versiones activas de Android (y iOS, fuera de este enfoque).
- **Personalización del fabricante**: ROMs modificadas, capas de personalización, restricciones de seguridad.
- **Conectividad variable**: 2G, 3G, 4G, 5G, WiFi, modo avión, etc.

### Tipos de pruebas relevantes en móvil

- **Pruebas funcionales**: validación de comportamiento esperado frente a requerimientos.
- **Pruebas de interfaz de usuario (UI):** validación de flujos e interacciones con el usuario.
- **Pruebas de compatibilidad**: en distintos dispositivos, versiones de Android y configuraciones regionales.
- **Pruebas de rendimiento**: tiempos de carga, uso de memoria, consumo de batería.
- **Pruebas de instalación y actualización**: verificación de procesos de instalación, desinstalación y upgrade.
- **Pruebas en segundo plano y multitarea**.

# Testing en aplicaciones nativas Android

## Arquitectura de una app Android

Las apps Android están compuestas por Activities, Fragments, Services, Content Providers y Broadcast Receivers, escritos en Java o Kotlin. El testing debe contemplar estas unidades de ejecución y sus interacciones.

## Tipos de pruebas en Android

- **Unitarias**: con JUnit, Mockito.
- **Instrumentadas (UI Tests):** con Espresso, UIAutomator.
- **Integración/End-to-End**: Appium, Robot Framework, Detox.

## Consideraciones específicas

- Manejo de permisos en tiempo de ejecución.
- Pruebas en emuladores vs. dispositivos reales.
- Testing en segundo plano (notificaciones, multitarea).

# Testing en aplicaciones React Native

## ¿Qué es React Native?

Framework multiplataforma basado en JavaScript/TypeScript que permite desarrollar aplicaciones móviles nativas para Android e iOS desde una única base de código, usando React.

## Arquitectura híbrida

React Native utiliza un "bridge" (puente) entre el código JavaScript y los módulos nativos del sistema operativo, lo que genera desafíos únicos para el testing.

## Tipos de pruebas aplicables

- Pruebas unitarias: Jest + React Testing Library (para componentes).
- Pruebas de integración: Enzyme, Testing Library.
- End-to-End: Detox (nativo), Appium (multiplataforma).

## Desafíos comunes

- Sincronización del "bridge" JavaScript-native.
- Pruebas en elementos asincrónicos (spinners, loaders).
- Flujos específicos por plataforma (Android vs. iOS).
- Acceso a componentes nativos desde JavaScript.

## Consideraciones para elegir herramientas de testing móvil

| Criterio | Android Native | React Native |
|---|---|---|
| **Lenguaje base** | Kotlin / Java | JavaScript / TypeScript |
| **Herramientas unitarias sugeridas** | JUnit, Mockito | Jest |
| **Herramientas UI/Funcionales** | Espresso, UIAutomator, Appium | Detox, Appium |
| **Automatización E2E** | Appium | Detox / Appium |
| **Integración CI/CD** | Gradle + Firebase / GitHub CI | Metro bundler + GitHub CI |
| **Pruebas en emuladores reales** | AVD, Genymotion, Firebase Test Lab | Android Emulator + Detox |

## Buenas prácticas en el testing de apps móviles

- Ejecutar pruebas en múltiples dispositivos reales y emuladores.
- Automatizar pruebas regresivas con herramientas como Appium.
- Simular condiciones reales (baja batería, pérdida de señal, rotación de pantalla).
- Hacer uso de capturas de pantalla automáticas al fallar un test.
- Integrar pruebas en pipelines CI/CD para feedback continuo.

# Capítulo 2: Instalación de APPIUM y configuración con emuladores

## ¿Qué es Appium?

Appium es una herramienta open source para la automatización de pruebas en aplicaciones móviles nativas, híbridas y web, sobre plataformas Android e iOS. Está basada en el protocolo WebDriver, lo que permite escribir pruebas en múltiples lenguajes (Java, Python, JavaScript, etc.).

- No requiere recompilación de la app.
- Compatible con múltiples lenguajes de automatización.
- Usa controladores específicos para cada plataforma (UIAutomator2 para Android).

## Requisitos previos para instalar Appium (entorno Android)

Antes de instalar Appium, se deben configurar las herramientas de desarrollo Android:

### Instalar Node.js y npm

Appium está basado en Node.js.

```bash
sudo apt install nodejs npm          # En Linux
brew install node                    # En macOS
```

Instalar Java JDK

```bash
sudo apt install openjdk-17-jdk     # Linux
```

Asegurarse de configurar la variable de entorno JAVA_HOME.

### Instalar Android Studio (incluye SDK, AVD y herramientas CLI)

- Descargar desde https://developer.android.com/studio
- Durante la instalación, seleccionar:
    - Android SDK
    - Android SDK Platform Tools
    - Android Virtual Device (AVD) Manager

### Configurar variables de entorno

Agregar en el archivo .bashrc, .zshrc o equivalente:

```bash
export ANDROID_HOME=$HOME/Android/Sdk
export PATH=$PATH:$ANDROID_HOME/emulator
export PATH=$PATH:$ANDROID_HOME/tools
export PATH=$PATH:$ANDROID_HOME/tools/bin
export PATH=$PATH:$ANDROID_HOME/platform-tools
```

## Instalación de Appium

### Instalar Appium Server

```bash
npm install -g appium
```

### Verificar instalación

```bash
appium -v
```

### Instalar Appium Doctor (opcional, para diagnóstico)

```bash
npm install -g appium-doctor
appium-doctor
```

Esto permite verificar que las dependencias de Appium estén correctamente instaladas.

## Configuración de emuladores Android (AVD)

### Crear un emulador con AVD Manager

Desde Android Studio:

- Ir a Tools > Device Manager > Create Device.
- Elegir un modelo (ej: Pixel 6).
- Seleccionar una imagen de sistema (ej: Android 12 x86_64 con Google APIs).
- Asignar nombre, resolución y recursos (RAM, almacenamiento).
- Finalizar y lanzar el emulador.

## Comandos CLI alternativos (avdmanager y emulator)

```bash
# Listar dispositivos disponibles
avdmanager list device

# Crear AVD desde terminal
avdmanager create avd -n my_avd -k "system-images;android-31;google_apis;x86_64"

# Ejecutar AVD
emulator -avd my_avd
```

## Verificar dispositivos activos

```bash
adb devices
```

## Configuración del entorno de pruebas con Appium

### Estructura mínima de un proyecto (Java + TestNG o JS + Mocha)

```bash
my-appium-project/
├── node_modules/
├── test/
│   └── sampleTest.js
├── package.json
├── app-debug.apk
└── capabilities.json
```

### Ejemplo básico de configuración de Desired Capabilities (JSON)

```json
{
  "platformName": "Android",
  "deviceName": "Pixel_6_API_31",
  "automationName": "UiAutomator2",
  "app": "/ruta/a/app-debug.apk",
  "appPackage": "com.ejemplo.miapp",
  "appActivity": ".MainActivity",
  "noReset": true
}
```

Estas capacidades se pasan al inicializar el driver de Appium desde el lenguaje elegido.

## Validación final

Una vez todo esté instalado y configurado:

Iniciar Appium Server:

```bash
appium
```

Ejecutar el emulador:

```bash
emulator -avd my_avd
```

Ejecutar pruebas desde tu framework (ej: WebDriverIO, Java + TestNG, Python + Pytest).

## Consideraciones avanzadas

- Se recomienda usar Appium Inspector para inspeccionar elementos UI en tiempo real.
- Es posible integrar Appium en pipelines CI/CD usando Docker o runners de GitHub Actions.
- Para pruebas paralelas o en múltiples dispositivos, se puede usar Appium Grid o herramientas como TestNG en paralelo.

# Capítulo 3: Simulación de gestos e interacciones

Las aplicaciones móviles requieren la validación de gestos táctiles complejos que son fundamentales para la experiencia de usuario: desplazamientos, toques prolongados, deslizamientos, doble toques, gestos de zoom, entre otros. Appium ofrece diversas formas de simular interacciones reales en dispositivos físicos o emuladores Android mediante su API basada en WebDriver y las capacidades del motor UiAutomator2.

## Principales gestos en testing móvil

| Gesto | Descripción técnica | Uso típico en app |
|---|---|---|
| **Tap (Toque)** | Simula un toque único sobre una coordenada o elemento | Presionar botón |
| **Long Press** | Toque sostenido por un período determinado | Mostrar menú |
| **Swipe (Deslizamiento)** | Movimiento en una dirección desde un punto inicial a uno final | Navegar en carruseles |
| **Scroll** | Similar a swipe, pero orientado a listas o contenido extensible | Buscar ítem |
| **Drag and Drop** | Arrastrar un elemento desde un origen hasta un destino | Reordenar elementos |
| **Pinch / Zoom** | Gestos con dos dedos: contracción o expansión | Mapas, imágenes |
| **Multi-Touch** | Acciones simultáneas con varios dedos | Juegos, gestos avanzados |

## Métodos de interacción en Appium

Appium permite realizar gestos mediante:

### Touch Actions (obsoletas en algunas versiones)

API clásica basada en acciones secuenciales:

```java
TouchAction touchAction = new TouchAction(driver);
touchAction
   .press(PointOption.point(500, 1000))
   .waitAction(WaitOptions.waitOptions(Duration.ofMillis(500)))
   .moveTo(PointOption.point(500, 500))
   .release()
   .perform();
```

### W3C Actions (actual estándar)

Permite la ejecución de gestos complejos y multi-touch de forma más flexible:

```javascript
await driver.performActions([
  {
    type: 'pointer',
    id: 'finger1',
    parameters: { pointerType: 'touch' },
    actions: [
      { type: 'pointerMove', duration: 0, x: 300, y: 1200 },
      { type: 'pointerDown', button: 0 },
      { type: 'pause', duration: 200 },
      { type: 'pointerMove', duration: 500, x: 300, y: 600 },
      { type: 'pointerUp', button: 0 }
    ]
  }
]);
```

## Ejemplos de gestos comunes

### Tap sobre un elemento

Java

```java
driver.findElement(By.id("com.example:id/btnLogin")).click();
```

JavaScript (WebDriverIO)

```javascript
await $('~btnLogin').click();
```

## Swipe (deslizar hacia arriba)

```javascript
await driver.performActions([
  {
    type: 'pointer',
    id: 'finger1',
    parameters: { pointerType: 'touch' },
    actions: [
      { type: 'pointerMove', duration: 0, x: 500, y: 1400 },
      { type: 'pointerDown', button: 0 },
      { type: 'pointerMove', duration: 500, x: 500, y: 400 },
      { type: 'pointerUp', button: 0 }
    ]
  }
]);
```

## Scroll hasta un elemento

Java

```java
MobileElement element = (MobileElement) driver.findElement(
  MobileBy.AndroidUIAutomator("new UiScrollable(new UiSelector().scrollable(true)).scrollIntoView(text(\"Aceptar\"));"));
element.click();
```

MobileElement element = (MobileElement) driver.findElement(
MobileBy.AndroidUIAutomator("new UiScrollable(new UiSelector().scrollable(true)).scrollIntoView(text(\"Aceptar\"));"));
element.click();

## Coordinadas absolutas vs relativas

Appium permite trabajar con:

- Coordenadas absolutas: píxeles fijos (menos portables entre dispositivos).
- Coordenadas relativas: cálculos dinámicos según tamaño de pantalla.

Ejemplo (JavaScript):

```javascript
const size = await driver.getWindowRect();
const startY = size.height * 0.8;
const endY = size.height * 0.2;
```

Requiere soporte del driver UiAutomator2 y dispositivo/emulador con multitouch habilitado.

## Buenas prácticas

- Verificar dimensiones de pantalla antes de definir coordenadas.
- Usar driver.getWindowRect() para adaptabilidad.
- Preferir interacciones por selector (id, accessibilityId) en vez de coordenadas fijas.
- Simular tiempos de espera realistas con pause o waitAction.
- Emplear Appium Inspector para capturar elementos y probar gestos visualmente.

# Capítulo 4: Validación de componentes móviles

La validación de componentes móviles consiste en verificar que los elementos visuales e interactivos de una aplicación —botones, campos de texto, listas, menús, notificaciones, etc.— funcionen de manera correcta y coherente con los requerimientos funcionales y de diseño. Appium, en conjunto con frameworks como WebDriverIO, TestNG o Pytest, permite realizar esta validación en entornos Android y React Native.

## ¿Qué se entiende por "componente móvil"?

Un componente móvil es cualquier elemento interactivo o visual de la interfaz de usuario. Entre los más comunes:

| Tipo de componente | Ejemplos comunes |
|---|---|
| **Elementos de entrada** | <EditText>, <TextInput>, <Switch> |
| **Botones** | <Button>, <TouchableOpacity> |
| **Navegación** | Menús, pestañas, drawer, toolbar |
| **Componentes de lista** | RecyclerView, FlatList |
| **Mensajes/feedback** | Toasts, snackbars, alertas, validaciones |
| **Multimedia** | Videos, imágenes, sliders |

## Estrategias de validación en Appium

Appium permite validar componentes a través de:

### Localización del elemento

- Por accessibilityId (recomendado para compatibilidad cross-platform).
- Por resource-id, class, xpath, text (Android).

Ejemplo (JavaScript / WebDriverIO):

```javascript
const loginButton = await $('~login_button');
await expect(loginButton).toBeDisplayed();
```

## Validaciones comunes

| Validación | Ejemplo de comando |
|---|---|
| **Elemento visible** | expect(element).toBeDisplayed() |
| **Elemento habilitado** | expect(element).toBeEnabled() |
| **Texto exacto o parcial** | expect(element).toHaveText('Bienvenido') |
| **Atributos (ej. placeholder, value)** | element.getAttribute('text') |
| **Estado lógico (checkbox, switch)** | element.getAttribute('checked') == 'true' |
| **Interacción (click, tap, input)** | element.setValue('correo@test.com') |

## Ejemplo en React Native

```javascript
const inputField = await $('~usernameInput');
await inputField.setValue('usuario123');
const loginBtn = await $('~submitButton');
await loginBtn.click();
```

Ejemplo en Android nativo

```java
MobileElement texto = driver.findElement(By.id("com.app:id/titulo"));
Assert.assertEquals("Hola mundo", texto.getText());
```

## Validación dinámica y asincronía

Las aplicaciones modernas suelen tener cargas asincrónicas, lo que exige validaciones basadas en espera explícita:

JavaScript (WebDriverIO)

```javascript
await browser.waitUntil(async () => {
  return (await $('~resultado')).isDisplayed();
}, { timeout: 5000, timeoutMsg: 'Resultado no visible' });
```

Java + Appium

```java
WebDriverWait wait = new WebDriverWait(driver, Duration.ofSeconds(10));
wait.until(ExpectedConditions.visibilityOfElementLocated(By.id("com.app:id/resultado")));
```

## Validación de listas (scroll + contenido)

En elementos como RecyclerView o FlatList, es común validar contenido escroleado:

```java
MobileElement item = (MobileElement) driver.findElement(
  MobileBy.AndroidUIAutomator("new UiScrollable(new UiSelector().scrollable(true))" +
  ".scrollIntoView(new UiSelector().text(\"Producto Final\"))"));
item.click();
```

## Buenas prácticas

- Nombrar componentes con accessibilityId en React Native (testID en JSX).
- Validar visibilidad, interactividad y contenido de cada componente esencial.
- Emplear aserciones claras con mensajes de error personalizados.
- Usar estructuras de Page Object Model para modularidad y reutilización.
- Capturar screenshots en caso de fallo para facilitar el debugging.

## Consideraciones específicas

- En apps React Native, el testID se mapea como accessibilityId.
- En Android nativo, preferir resource-id sobre xpath para estabilidad.

Validar también comportamiento negativo, como errores al dejar campos vacíos o presionar botones sin completar el formulario.

# Capítulo 5: Exploración de DETOX como alternativa para REACT NATIVE

## ¿Qué es Detox?

Detox es un framework end-to-end (E2E) de código abierto diseñado específicamente para aplicaciones móviles desarrolladas en React Native. Fue creado por Wix con el objetivo de permitir pruebas funcionales confiables, sincronizadas y de alto rendimiento sobre componentes nativos e híbridos, usando JavaScript/TypeScript y ejecutando las pruebas en dispositivos reales o emuladores.

## ¿Por qué considerar Detox como alternativa a Appium en React Native?

| Característica | Appium | Detox |
|---|---|---|
| **Soporte multiplataforma** | Android / iOS / apps híbridas / web | Android / iOS (solo React Native) |
| **Lenguaje** | Multilenguaje (Java, Python, JS, etc.) | JavaScript / TypeScript |
| **Configuración** | Compleja, especialmente en CI | Sencilla, integrada al entorno RN |
| **Velocidad** | Moderada | Alta (por ejecución nativa) |
| **Sincronización automática** | Limitada (necesita esperas explícitas) | Sí (detecta idle state automáticamente) |
| **Nivel de integración** | Externo a la app | Interno (compila con la app en modo test) |
| **Inspección visual** | Appium Inspector | No incluye inspector visual |

Detox es una alternativa ligera, rápida y específica para React Native, ideal cuando se trabaja exclusivamente con este framework y se requiere integración CI/CD fluida.

## Arquitectura de Detox

Detox trabaja desde dentro de la aplicación, compila una versión especial de la app con su propio runtime de pruebas y ejecuta comandos en sincronía con la ejecución nativa, lo que elimina la necesidad de esperas manuales (wait o pause).

Componentes principales:

- detox-cli: herramienta de línea de comandos.
- jest (u otro runner): ejecutor de pruebas.
- Instrumentación: inyección de código en la app RN.

## Instalación y configuración básica

### Requisitos

- Node.js ≥ 14
- React Native ≥ 0.63
- Android Studio y/o Xcode (según plataforma)
- Emulador configurado
- Yarn o npm

### Instalación del framework Detox

```bash
npm install detox --save-dev
npm install jest --save-dev
detox init -r jest
```

Esto genera:

- e2e/ carpeta de pruebas.
- package.json con scripts y configuración.
- Archivo detox.config.js.

**Configuración del detox.config.js**

```js
module.exports = {
  testRunner: 'jest',
  runnerConfig: 'e2e/config.json',
  configurations: {
    "android.emu.debug": {
      type: "android.emulator",
      device: {
        avdName: "Pixel_4_API_30"
      },
      app: "android/app/build/outputs/apk/debug/app-debug.apk"
    }
  }
};
```

## Ejemplo de prueba con Detox

```javascript
describe('Login flow', () => {
  beforeAll(async () => {
    await device.launchApp();
  });

  it('should show login screen', async () => {
    await expect(element(by.id('loginScreen'))).toBeVisible();
  });

  it('should login successfully', async () => {
    await element(by.id('username')).typeText('testuser');
    await element(by.id('password')).typeText('123456');
    await element(by.id('loginButton')).tap();
    await expect(element(by.id('homeScreen'))).toBeVisible();
  });
});
```

## Ventajas técnicas de Detox

- Sincronización automática con la UI y JavaScript: evita fallos por tiempos de carga variables.
- Rendimiento superior a herramientas basadas en WebDriver.
- Menor mantenimiento de pruebas debido a su acoplamiento con el ciclo de vida de la app.
- Fácil integración en CI/CD con GitHub Actions, CircleCI, Bitrise, etc.

## Limitaciones y consideraciones

- Solo compatible con React Native (y apps RN-compatibles).
- Requiere compilar una versión especial de la app con código Detox embebido.
- No permite pruebas cross-platform desde un mismo script (a diferencia de Appium).
- Carece de inspección visual como Appium Inspector (aunque se puede combinar con herramientas como Flipper para debugging).

## Casos de uso ideales para Detox

- Validar flujos completos en React Native con alta precisión.
- Pruebas rápidas en pipelines CI.
- Verificación de integración entre componentes (pantallas, navegación, formularios).
- Validación frecuente en proyectos activos donde el tiempo de ejecución importa.