Propuesta de arquitectura para framework de automatización mobile

> **Objetivo:** Definir una arquitectura escalable, mantenible y multiplataforma (Android + iOS) para pruebas automatizadas usando **Java 11+**, **Appium**, patrón **POM** y **JUnit/TestNG**. Dependencias gestionadas con **Maven/Gradle** e integración de logging con **SLF4J + Log4j2**.

---

1. Visión general

El framework está diseñado para automatizar pruebas en aplicaciones móviles de tres tipos: - **Nativas:** desarrolladas específicamente para Android o iOS utilizando sus SDKs. - **Web móviles:** aplicaciones accesibles desde el navegador del dispositivo. - **Híbridas:** aplicaciones que combinan elementos nativos con contenido web incrustado en un WebView (ejemplo: apps hechas con Ionic, Cordova o React Native). En este caso, el framework debe ser capaz de cambiar de contexto entre la capa nativa y la capa web para validar correctamente la funcionalidad.

El objetivo es maximizar la reutilización de código entre Android e iOS, garantizando una arquitectura modular, basada en separación de responsabilidades y en el uso de patrones de diseño que faciliten el mantenimiento y la escalabilidad.

---

2. Estructura del proyecto

La organización del proyecto sigue una estructura clara para separar responsabilidades y facilitar el mantenimiento:

```
mobile-automation-framework/
├── pom.xml (o build.gradle)        # Archivo principal para gestionar
dependencias y configuración del build
├── src/
│   ├── main/java/com.company.framework/
│   │   ├── config/                 # Clases para leer y manejar
configuraciones, perfiles y factories
│   │   ├── drivers/                # DriverFactory y clases relacionadas con
la creación de drivers Android/iOS
│   │   ├── pages/                  # Page Objects que representan pantallas y
componentes de la app
│   │   ├── services/               # Clases que encapsulan flujos de negocio
reutilizables (login, compra, etc.)
│   │   ├── utils/                  # Funciones utilitarias (esperas
explícitas, manejo de gestos, helpers)
```

```
│   │   └── reporting/              # Integración con librerías de reportes
(Allure, ExtentReports)
│   └── test/java/com.company.tests/
│       ├── suites/                 # Definición de suites de pruebas (smoke,
regression, integración)
│       └── tests/                  # Casos de prueba individuales organizados
por funcionalidad
├── resources/                      # Archivos de configuración (propiedades,
JSON de capabilities, log4j2)
└── reports/                        # Carpeta donde se generan los reportes
tras cada ejecución
```

Esta estructura garantiza que: - `main` contiene el core del framework (código común, drivers, utilidades, configuración). - `test` contiene los casos de prueba y las suites, separados del core. - `resources` centraliza la configuración para distintos entornos y plataformas. - `reports` almacena resultados de ejecución, facilitando la trazabilidad en CI/CD.

---

3. Patrones aplicados

- **Page Object Model (POM):** Cada pantalla o componente de la aplicación se modela como una clase que encapsula los localizadores y las acciones disponibles. Esto centraliza la lógica de interacción con la UI, reduce duplicación de código y facilita el mantenimiento cuando cambian los elementos de la interfaz.

- **Factory:** Se utiliza para crear instancias del driver de Appium según la plataforma objetivo (AndroidDriver o IOSDriver). El patrón abstrae la inicialización de los drivers y permite intercambiar entornos de ejecución (local, remoto, emulador, dispositivo físico) sin modificar el código de prueba.

- **Strategy/Builder:** Facilita el manejo de configuraciones dinámicas. Strategy permite seleccionar diferentes fuentes de datos (archivos JSON, propiedades, variables de entorno), mientras que Builder se encarga de combinar estas fuentes y construir las DesiredCapabilities necesarias para inicializar los tests.

- **Singleton/ThreadLocal:** El Singleton garantiza que la configuración global se cree una sola vez. ThreadLocal asegura que, en ejecuciones en paralelo, cada hilo de ejecución tenga su propia instancia de AppiumDriver, evitando interferencias entre pruebas y asegurando aislamiento.

---

4. Configuración multiplataforma

La configuración se maneja de manera flexible para soportar Android e iOS sin duplicar esfuerzos:

- **Archivos de propiedades y JSON:** Contienen la configuración base por plataforma (ejemplo: `android.conf.json`, `ios.conf.json`) con las *DesiredCapabilities* necesarias.

- **Perfiles y variables de entorno:** Mediante perfiles de Maven/Gradle y el uso de variables de entorno (ejemplo: `PLATFORM=android`, `ENV=ci`) se controla en qué plataforma y entorno se ejecutan las pruebas (local, remoto, CI/CD).

- **Combinación dinámica:** Un gestor central de configuración lee tanto los archivos base como las variables de entorno/perfiles y construye la configuración final que se usará para inicializar el driver y ejecutar los tests.

5. Logs y reportes

- **Logging:** Se implementa con SLF4J + Log4j2, lo que permite unificar el manejo de logs, personalizar el nivel de detalle (INFO, DEBUG, ERROR) y generar salidas tanto en consola como en archivos separados por ejecución.
- **Reportes:** Se recomienda usar Allure o ExtentReports para centralizar resultados, incluir evidencias como capturas de pantalla, logs de dispositivo o incluso videos, y facilitar la integración con herramientas de CI/CD para que los reportes estén siempre disponibles tras la ejecución.

---

6. Escalabilidad y mantenibilidad

- **Arquitectura modular:** El framework se organiza en capas (core, pages, tests) para que cada equipo trabaje de forma independiente y el mantenimiento sea más sencillo.
- **Reutilización de flujos:** Se definen clases de servicio o flows que combinan acciones de varias páginas, evitando duplicar lógica en los tests.
- **Integración con CI/CD:** Los pipelines permiten ejecutar diferentes tipos de suites (smoke, regression, integration) según la necesidad del proyecto.
- **Documentación y versionamiento:** Una buena documentación y el uso de versionamiento semántico facilitan la evolución del framework y evitan incompatibilidades.

---

7. Organización de dependencias y ejecución

- **Gestión de dependencias:** Se utilizan Maven o Gradle para mantener versiones controladas y reproducibles de todas las librerías.
- **Perfiles por plataforma y entorno:** Los perfiles de ejecución (`-Pandroid`, `-Pios`, `-Plocal`, `-Premote`) simplifican el cambio entre distintos escenarios.
- **Agrupación de pruebas:** Los casos se organizan en suites (smoke, regression, integration) para ejecutar solo lo necesario en cada momento.
- **Ejecución en paralelo:** El framework soporta pruebas concurrentes, tanto en entornos locales como en plataformas en la nube (BrowserStack, Sauce Labs, etc.).

---

8. Recomendaciones

- **Allure como estándar de reportes:** Permite trazabilidad completa de pruebas y adjuntar evidencias.
- **Modularización progresiva:** Si el proyecto crece, separar en módulos específicos (core, android, ios, tests) asegura escalabilidad.
- **Convenciones claras:** Definir estándares de nombres, estructura de carpetas y estrategia de branching asegura orden y consistencia a largo plazo.