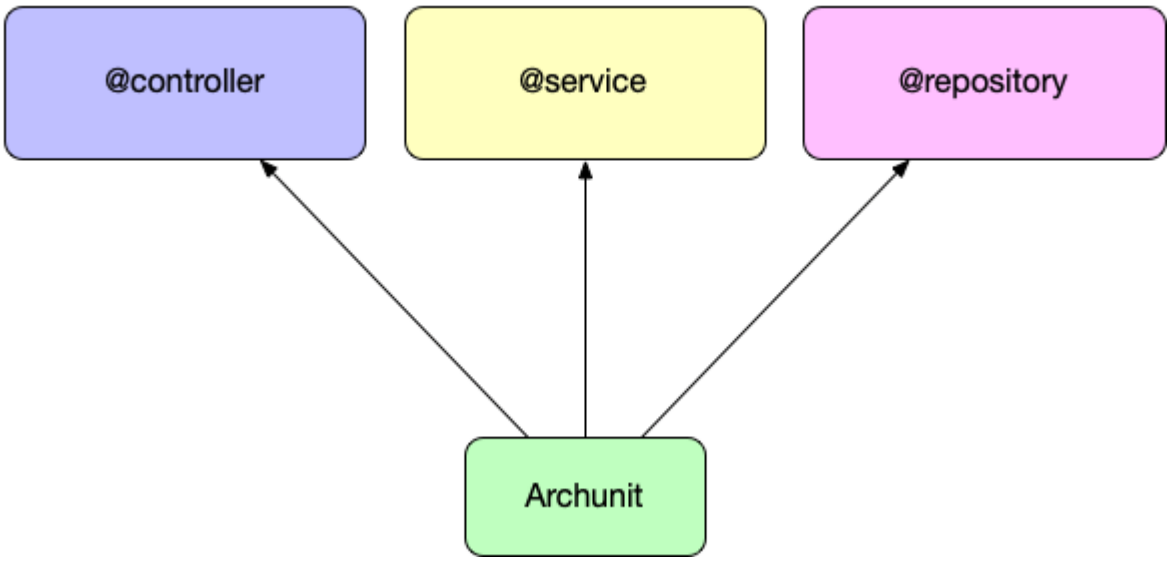
1. Introducción: por qué merece la pena hablar de ArchUnit

Seguro que te ha pasado alguna vez: empiezas una aplicación Java con una arquitectura clara, paquetes bien separados, capas con responsabilidades definidas y una idea bastante limpia de cómo debe crecer el proyecto. Pero pasan los sprints, llegan nuevas funcionalidades, aparecen urgencias y, poco a poco, alguien desde un `controller` llama directamente a un `repository`, una clase de dominio empieza a depender de detalles técnicos o los nombres dejan de seguir la convención acordada.

El problema no suele ser falta de criterio. Muchas veces, simplemente no hay nada automático que proteja esas decisiones de arquitectura cuando el código cambia.

Ahí es donde entra ArchUnit, una librería que permite escribir tests para validar reglas de arquitectura en proyectos Java. Con ArchUnit podemos comprobar dependencias entre paquetes, capas, nombres de clases y buenas prácticas del diseño, igual que comprobamos comportamiento con tests unitarios o de integración.



Y esa es la idea interesante: si como desarrolladores ya aceptamos que el código necesita

tests, también tiene sentido que la arquitectura tenga los suyos. No para burocratizar el proyecto, sino para detectar desviaciones pronto, antes de que se conviertan en deuda difícil de corregir.

A lo largo del artículo veremos ejemplos sencillos usando el package `com.arquitecturajava`, con una estructura fácil de adaptar a un proyecto real.

2. Qué es ArchUnit y qué problemas resuelve

ArchUnit es una librería Java pensada para analizar el bytecode de nuestras clases y comprobar si cumplen ciertas reglas de arquitectura. Lo interesante es que esas reglas se escriben como tests automatizados, así que pueden ejecutarse junto al resto de la suite cada vez que lanzamos los tests del proyecto.

En la práctica, ArchUnit ayuda a detectar problemas muy habituales antes de que se mezclen demasiado con el código. Por ejemplo:

- Evitar que una clase de `controller` acceda directamente a `repository`, saltándose la capa de `service`.
- Impedir dependencias circulares entre paquetes o módulos.
- Comprobar que las clases terminadas en `Controller`, `Service` o `Repository` estén en el paquete correcto.
- Validar que ciertos servicios estén anotados correctamente, por ejemplo con `@Service` si usamos Spring.
  Una de sus ventajas es que se integra muy bien con JUnit. No hace falta aprender una herramienta externa ni montar un proceso complejo en el pipeline. Creamos una clase de test, definimos reglas y dejamos que ArchUnit las verifique.

Dicho de forma directa: ArchUnit en Java permite convertir decisiones de arquitectura en reglas ejecutables dentro de la suite de tests. Eso no sustituye el criterio del equipo, ni una buena revisión de código, pero sí actúa como una red de seguridad para detectar

desviaciones antes de llegar a producción.

3. Primer ejemplo con ArchUnit en el package com.arquitecturajava

Imaginemos una estructura sencilla para el proyecto:

```
com.arquitecturajava.controller
com.arquitecturajava.service
com.arquitecturajava.repository
com.arquitecturajava.domain
```

La idea será validar una regla muy típica: un `controller` puede llamar a `service`, pero no debería acceder directamente a `repository`.

Para empezar, añadimos ArchUnit junto con JUnit 5. En Maven sería algo así:

```
<dependency>
    <groupId>org.example</groupId>
    <artifactId>nombre-libreria</artifactId>
    <version>1.0.0</version>
</dependency>
```

O en Gradle:

```
testImplementation 'com.tngtech.archunit:archunit-junit5:1.3.0'
```

Ahora creamos una clase de test llamada `ArquitecturaTest`, por ejemplo dentro de `com.arquitecturajava`:

```
package com.arquitecturajava;

import com.tngtech.archunit.junit.AnalyzeClasses;
```

```
import com.tngtech.archunit.junit.ArchTest;
import com.tngtech.archunit.lang.ArchRule;

import static
com.tngtech.archunit.library.Architectures.layeredArchitecture;

@AnalyzeClasses(packages = "com.arquitecturajava")
class ArquitecturaTest {

    @ArchTest
    static final ArchRule controller_solo_accede_a_service =
        layeredArchitecture()
            .consideringOnlyDependenciesInLayers()
            .layer("Controller").definedBy("..controller..")
            .layer("Service").definedBy("..service..")
            .layer("Repository").definedBy("..repository..")
            .whereLayer("Controller").mayOnlyAccessLayers("Service");
}
```

¿Qué está pasando aquí? ArchUnit carga las clases del paquete `com.arquitecturajava`, detecta las capas según los paquetes y aplica la regla. Si mañana alguien modifica un `PedidoController` para inyectar directamente un `PedidoRepository`, el test fallará.

Lo interesante es que no estamos escribiendo una comprobación manual ni una convención en un documento perdido. Estamos convirtiendo una decisión de arquitectura en un test ejecutable, igual que hacemos con la lógica de negocio.

4. Reglas útiles de ArchUnit para proyectos Java reales

Una buena forma de empezar con ArchUnit es elegir reglas pequeñas, fáciles de entender y

con impacto real. No hace falta cubrir toda la arquitectura el primer día. Mejor empezar por convenciones claras y hacer que el equipo las vea como una ayuda, no como una barrera.

Por ejemplo, podemos asegurar que las clases que terminan en `Controller` estén en su paquete correcto:

```
@ArchTest
static final ArchRule controllers_deben_estar_en_controller =
    classes()
        .that().haveSimpleNameEndingWith("Controller")
.should().resideInAPackage("com.arquitecturajava.controller..");
```

También podemos aplicar una regla similar para los servicios. Si usamos Spring, además podemos exigir la anotación `@Service`:

```
@ArchTest
static final ArchRule services_deben_estar_bien_definidos =
    classes()
        .that().haveSimpleNameEndingWith("Service")
        .should().resideInAPackage("com.arquitecturajava.service..")
        .andShould().beAnnotatedWith(Service.class);
```

Otra regla muy útil es proteger el dominio. Si queremos un modelo más limpio, `domain` no debería depender de capas externas ni de frameworks:

```
@ArchTest
static final ArchRule domain_no_debe_depender_de_capas_externas =
    noClasses()
        .that().resideInAPackage("com.arquitecturajava.domain..")
        .should().dependOnClassesThat().resideInAnyPackage(
            "com.arquitecturajava.controller..",
            "com.arquitecturajava.repository..",
```

```
            "org.springframework.."
        );
```

Y para reforzar la dirección de dependencias, podemos impedir que `repository` conozca `controller`:

```
@ArchTest
static final ArchRule repository_no_debe_depender_de_controller =
    noClasses()
        .that().resideInAPackage("com.arquitecturajava.repository..")
        .should().dependOnClassesThat()
        .resideInAPackage("com.arquitecturajava.controller..");
```

El consejo práctico es sencillo: empieza con reglas simples, explícales el motivo al equipo y evita normas demasiado rígidas. Las reglas de arquitectura en Java funcionan mejor cuando protegen decisiones importantes, no cuando generan fricción innecesaria.

5. Cómo introducir ArchUnit en un equipo sin complicarse

ArchUnit suele funcionar mejor cuando entra poco a poco, sobre todo en proyectos existentes donde ya hay cierta deuda técnica. Si el primer día intentamos validar toda la arquitectura, lo normal es encontrarnos con demasiados fallos y generar rechazo. Es más útil empezar con una regla clara, fácil de entender y alineada con una decisión importante del equipo.

Un buen primer paso es documentar la arquitectura actual, aunque sea en una página sencilla:

- Qué paquetes existen, por ejemplo `com.arquitecturajava.controller`, `service`, `repository` y `domain`.
- Qué capas pueden hablar entre sí.

- Qué dependencias están permitidas.
- Qué dependencias queremos evitar a partir de ahora.
  Después podemos crear una primera batería de tests en un paquete como `com.arquitecturajava.architecture` o `com.arquitecturajava.test`, según la convención del proyecto. Lo importante es que esos tests sean visibles, se ejecuten en la suite habitual y formen parte del flujo normal de desarrollo.

  También conviene que cada regla nueva se añada con consenso. Si el equipo entiende el motivo, ArchUnit deja de parecer una imposición y se convierte en una red de seguridad compartida.

  En proyectos legacy, algunas reglas pueden marcarse como objetivo futuro. Por ejemplo, se pueden revisar en local o en CI sin bloquear el pipeline desde el primer día.

  Al hablar de estas prácticas, es natural reforzar ideas como tests de arquitectura con ArchUnit, reglas de arquitectura en Java y validar capas con ArchUnit, porque describen justo el valor que aporta la herramienta.

6. Conclusión: ArchUnit como red de seguridad para tu arquitectura

ArchUnit encaja muy bien como una red de seguridad para la arquitectura de una aplicación Java. Igual que un test unitario nos avisa cuando rompemos una lógica concreta, un test de arquitectura nos avisa cuando el diseño empieza a desviarse de lo acordado por el equipo.

Los ejemplos con `com.arquitecturajava` muestran una idea sencilla pero muy potente: podemos convertir reglas reales sobre capas como `controller`, `service`, `repository` y `domain` en tests claros, automatizados y fáciles de ejecutar. No hace falta esperar a una revisión manual para detectar que un `controller` está accediendo directamente a un `repository`, o que el dominio empieza a depender de detalles que no debería conocer.

La mejor forma de empezar no es cubrir toda la arquitectura de golpe. Prueba ArchUnit con una sola regla, por ejemplo: impedir que `com.arquitecturajava.controller` dependa de `com.arquitecturajava.repository`. Si esa regla aporta valor y el equipo la entiende, añade otra.

Al final, no se trata de llenar el proyecto de normas ni de bloquear el trabajo diario. Se trata de cuidar las decisiones importantes para que el código siga siendo mantenible con el paso del tiempo.

Revisa hoy la arquitectura actual de tu proyecto, elige una regla sencilla y conviértela en un test con ArchUnit. Esa pequeña decisión puede evitar muchos problemas mañana.

- Maven : ¿Que es? .¿Como funciona?
- La regla del tres y la refactorización
- Maven Tutorial , introducción a Maven
- Curso Diseño de Base de Datos y modelado
- Curso Spring Data y buenas prácticas