

Desarrollo de Microservicios Cloud-Native con Spring Boot

Arquitectura, Persistencia y Compilación Nativa en el Ecosistema Moderno

Los Tres Pilares del Ciclo de Vida Moderno

Cimientos y Configuración



Cimientos y Configuración

Inyección de dependencias y configuración externalizada ([Spring Core & Web](#)).

Estado y Datos



Estado y Datos

Persistencia declarativa y transaccionalidad ([Spring Data JPA](#)).

Optimización AOT



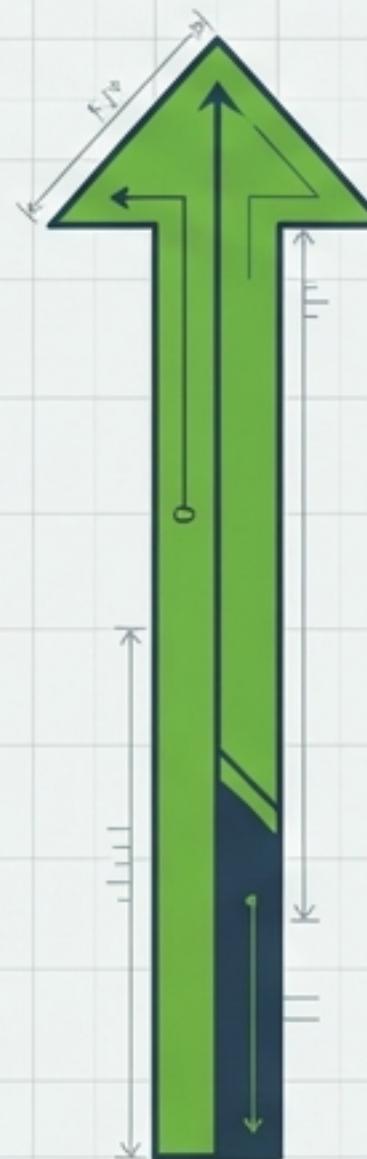
Optimización AOT

Compilación nativa y eficiencia de recursos ([Spring Boot AOT & GraalVM](#)).

Configuración Externalizada: La Jerarquía de Precedencia

Spring Boot prioriza la configuración externa sobre la interna.

Mayor Precedencia



Command Line Arguments

Java System Properties (-D)

OS Environment Variables

Application Properties
(application.yml)

Menor Precedencia

Entorno (Runtime)

Código / Empaquetado

Inyección de Dependencias y Scopes

Concepto	Anotación Spring	Comportamiento
Singleton	@Component / @Service	Una sola instancia por aplicación (Default).
Request	@RequestScope	Una nueva instancia por cada petición HTTP.
Prototype	@Scope("prototype")	Una nueva instancia cada vez que se inyecta.

Construyendo APIs RESTful con Spring Web MVC

```
1 @RestController
2 @RequestMapping("/api/usuarios")
3 public class UsuarioController {
4
5     @GetMapping("/{id}")
6     public Usuario getUsuario(@PathVariable("id") String id) {
7         return servicio.buscarPorId(id);
8     }
9 }
```

Equivalencias

- **@RestController**
 - Define el componente Web
- **@RequestMapping**
 - Ruta base del recurso
- **@GetMapping**
 - Mapea peticiones HTTP GET
- **@PathVariable**
 - Extrae valores de la URL

Arquitectura de Persistencia: Spring Data JPA

Eliminando el código repetitivo (Boilerplate)

```
1 public class EmployeeService {  
2     @Inject  
3     EntityManager em;  
4  
5     @Transactional  
6     public void createEmployee(String name) {  
7         Employee employee = new Employee();  
8         employee.setName(name);  
9         em.persist(employee);  
10    }  
11 }
```

Gestión Manual (EntityManager)

Abstracción Automática

```
1 @Entity  
2 public class Expense {  
3     @Id  
4     @GeneratedValue  
5     private Long id;  
6     private String name;  
7  
8 }
```

Entidad JPA Estándar

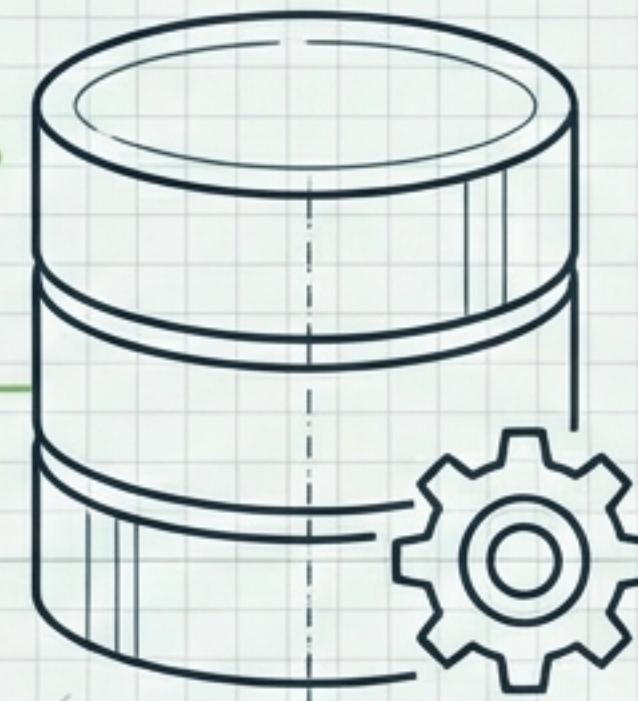
El Patrón Repositorio

Interfaces declarativas en lugar de implementaciones manuales.

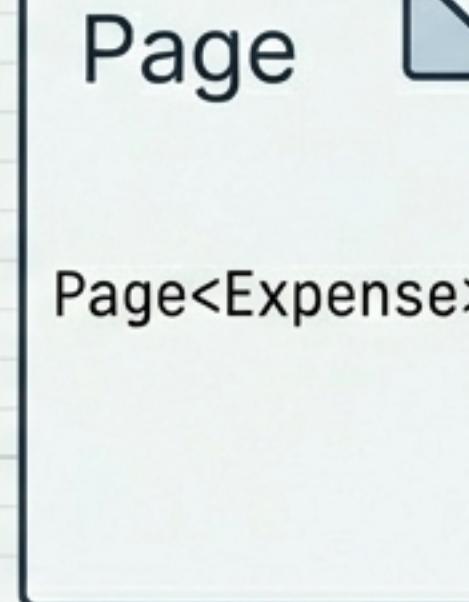
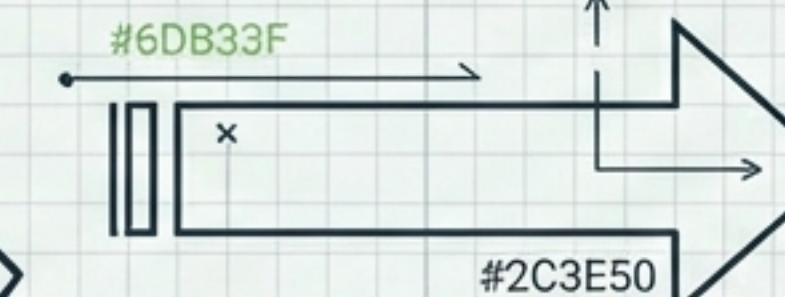
```
@Repository  
public interface ExpenseRepository extends JpaRepository<Expense, Long> {  
  
    // "Magic Method" - Spring genera el SQL automáticamente  
    List<Expense> findByName(String name);  
}
```

Query Method: **Generación automática**
de consultas basada en el nombre del
método.

Consultas Avanzadas: Paginación y Ordenamiento



Base de Datos
(10,000 registros)



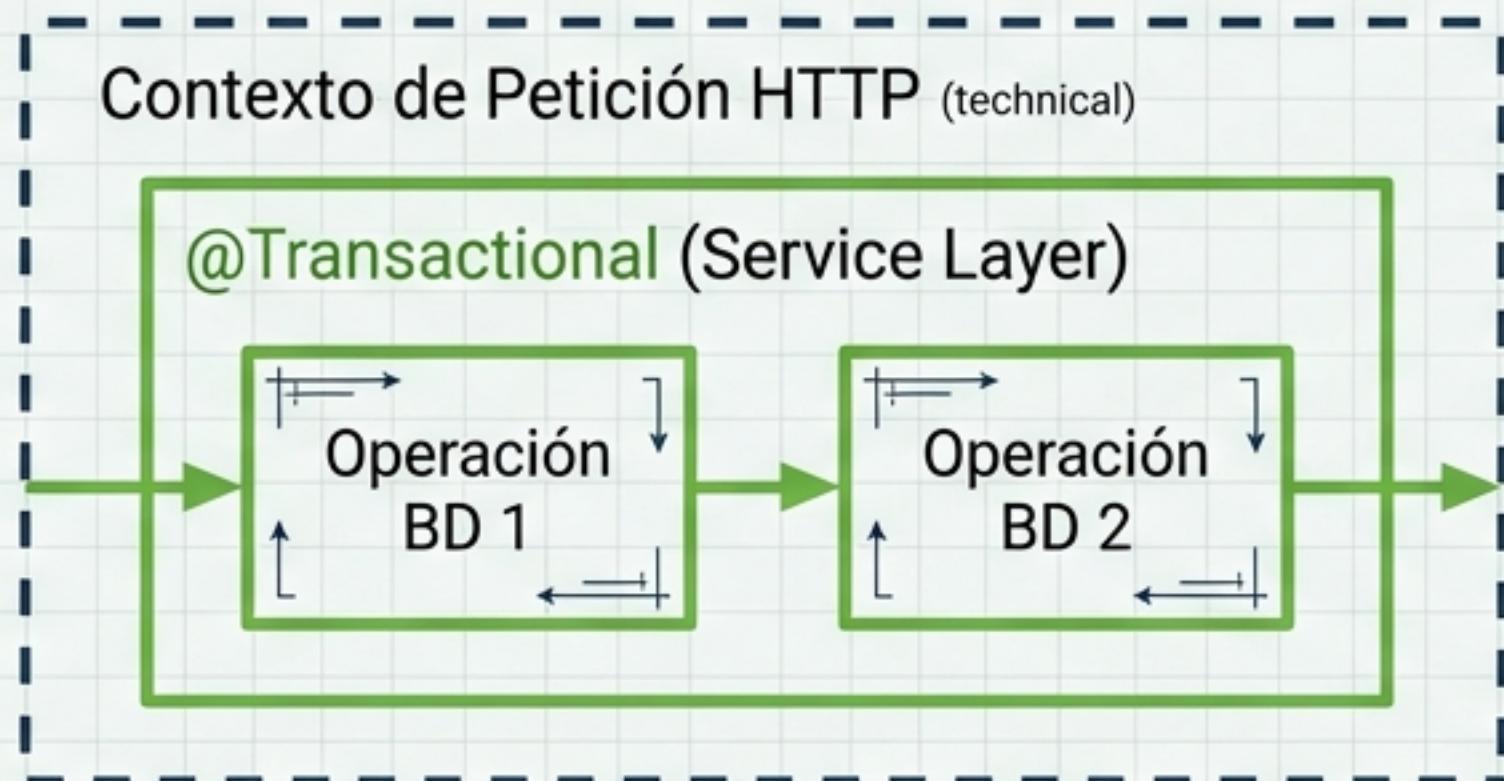
Ordenamiento

```
// Página 0, Tamaño 10  
Pageable page = PageRequest.of(0, 10);
```

```
// Ordenar por nombre ascendente  
Sort sort = Sort.by("name").ascending();
```

Gestión Declarativa de Transacciones

Control de atomicidad con @Transactional



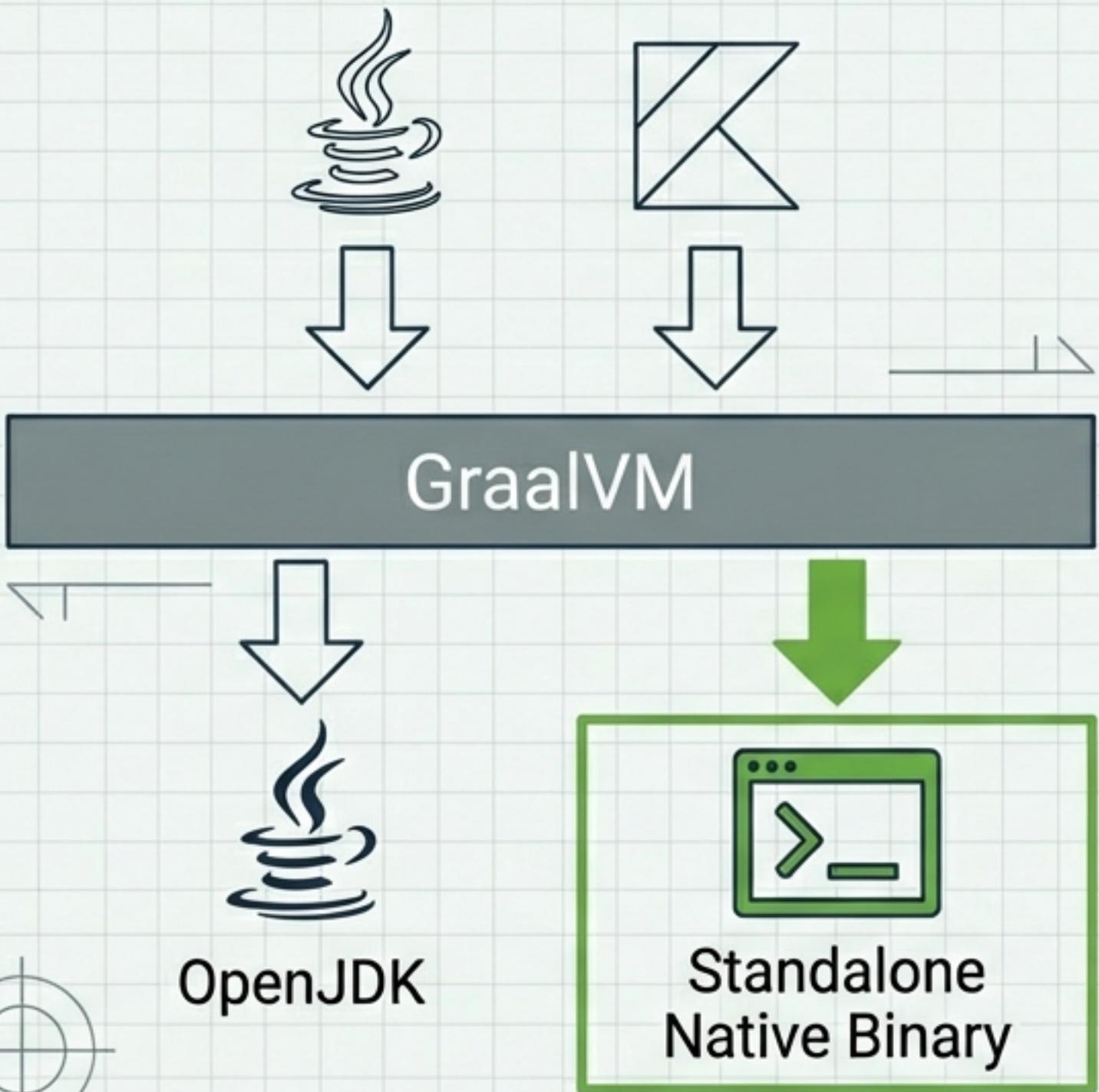
Tipos de Propagación

- **REQUIRED** (Default):
Se une a la transacción actual o crea una nueva.
 - **REQUIRES_NEW**:
Suspende la actual y crea una transacción independiente.
 - **MANDATORY**:
Falla si no existe una transacción activa.

```
Blueprint Technic: JetBrains Mono

@Transactional( TxType.REQUIRES_NEW )
public void createEmployee( String name ) {
    Employee employee = new Employee();
    employee.setName( name );
    em.persist( employee );
}
```

La Revolución Nativa: Spring Boot + GraalVM



JVM (JIT)

- Arranque: Lento (Segundos)
- Memoria: Alta
- Optimización: Dinámica

GraalVM (AOT)

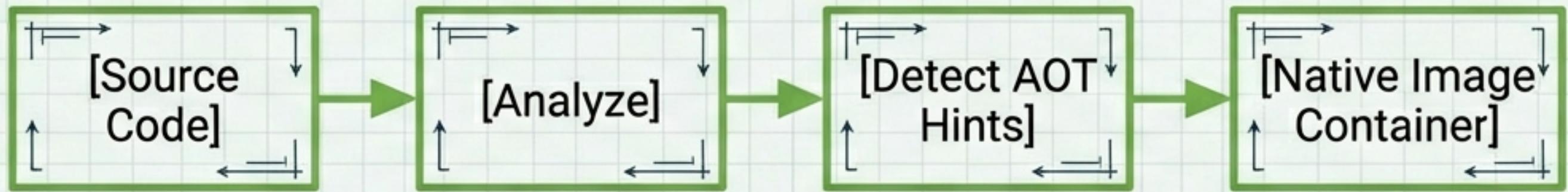
- Arranque: Instantáneo (Milisegundos)
- Memoria: Baja
- Optimización: Estática (Build-time)

Construcción de Imágenes Nativas (AOT)

Del código fuente al contenedor sin Dockerfiles manuales.

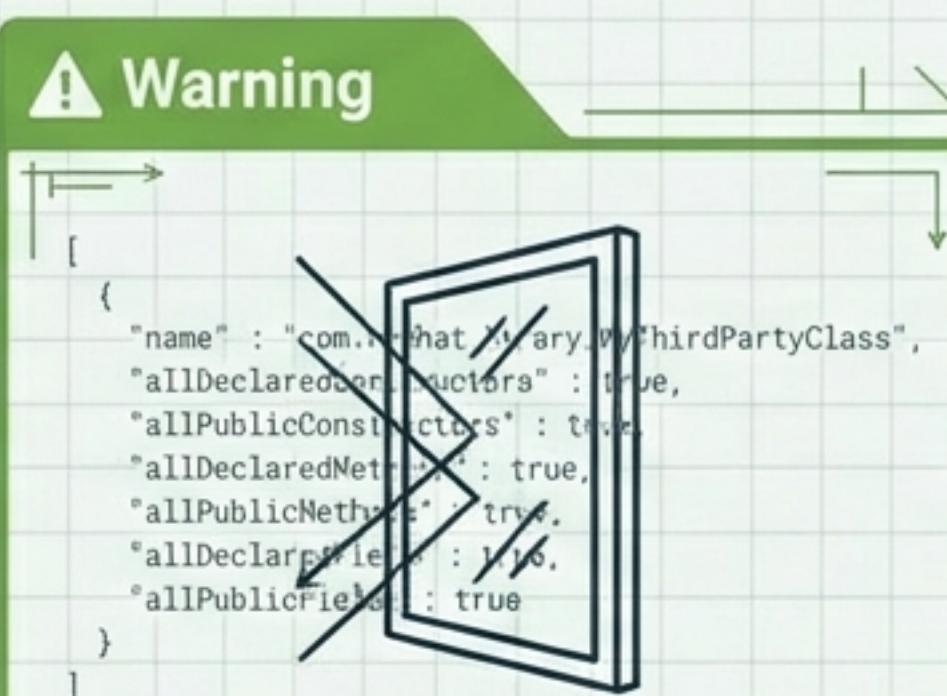
```
$ mvn -Pnative spring-boot:build-image
```

Cloud Native Buildpacks



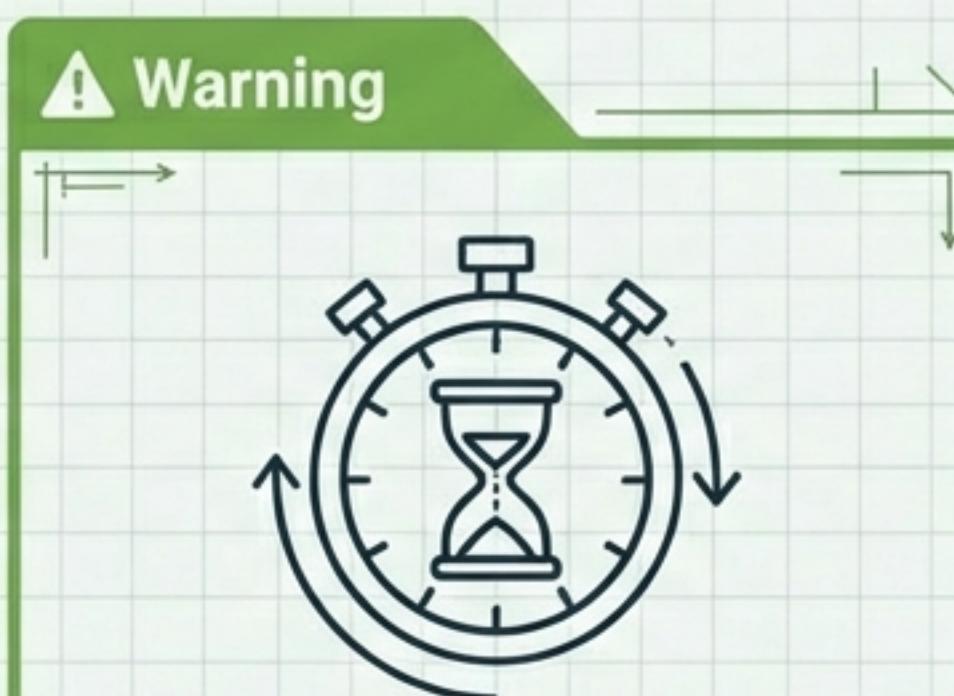
Desafíos y Limitaciones de AOT

La suposición del "Mundo Cerrado"



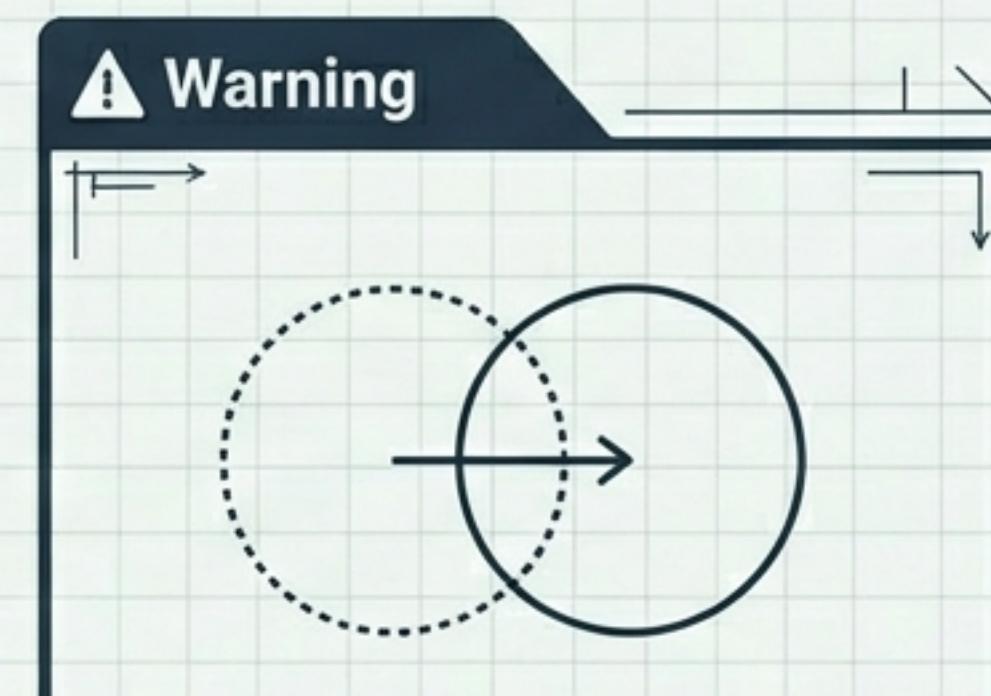
Reflexión (Reflection)

GraalVM necesita conocer todas las clases dinámicas al compilar. Requiere "Runtime Hints" explícitos.



Tiempo de Build

La compilación nativa consume mucha CPU y RAM. Puede tardar minutos en lugar de segundos.



Proxies Dinámicos

Limitado. Spring AOT pre-calcula los proxies necesarios durante la fase de construcción.



¿Cuándo usar Imágenes Nativas?

TECHNICAL MANIFEST: NATIVE IMAGE USE CASES

Arquitecturas Serverless (AWS Lambda, Knative)

Inicio rápido y bajo consumo de memoria son críticos.



Entornos de Alta Densidad (Kubernetes)

Maximiza la utilización de recursos en pods.



Herramientas de Línea de Comandos (CLI)

Ejecución instantánea para usuarios.



Arranque Instantáneo Requerido

Elimina el tiempo de calentamiento de la JVM.

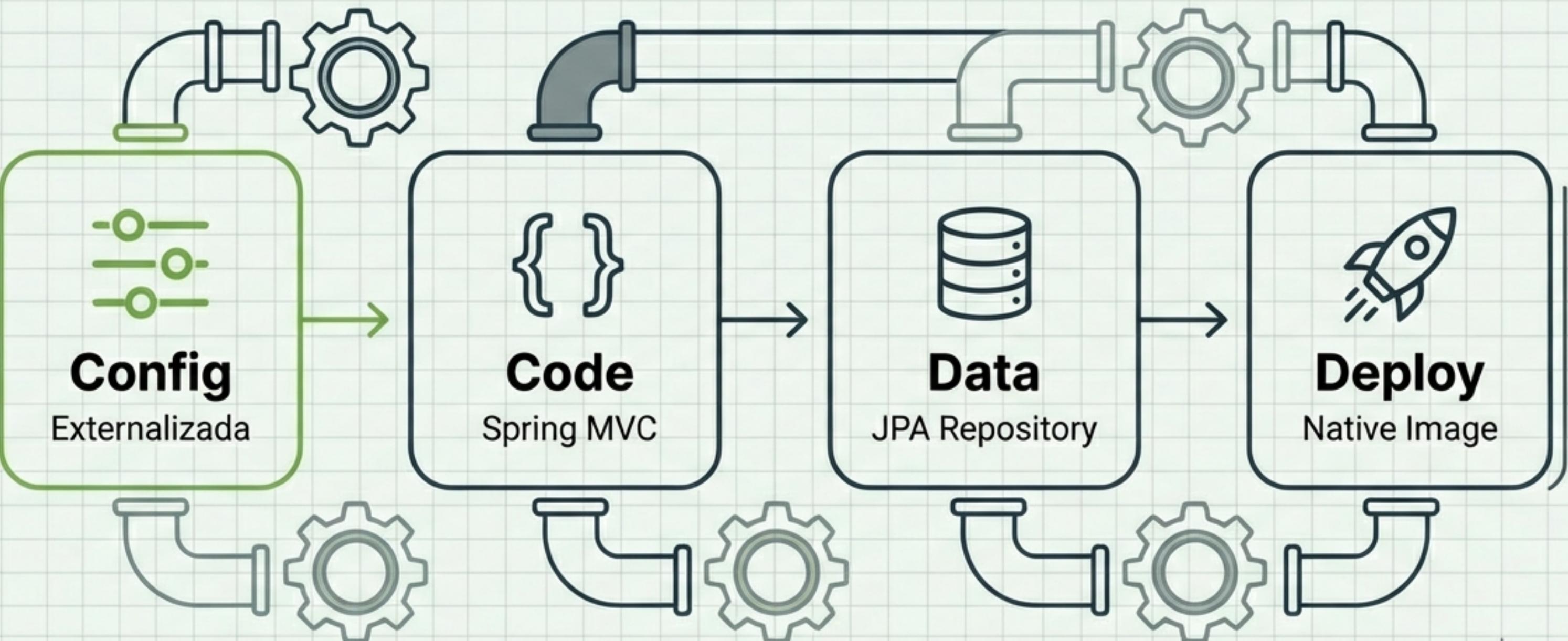


Monolitos de Larga Duración (JIT puede ser más rápido en throughput)

La optimización en tiempo de ejecución de JIT puede superar el rendimiento nativo a largo plazo.



Resumen del Stack Cloud-Native



La eficiencia del desarrollo se une al rendimiento de la ejecución.

Recursos y Siguientes Pasos

- Spring Initializr
(start.spring.io)
- Spring Boot 3.4 Reference Documentation
- GraalVM Native Image Guide
- Spring Data JPA Repositories

Documentación Oficial



Referencias basadas en la documentación oficial de Spring y GraalVM.