Universidad Icesi

Spring Framework: Anotaciones, JavaConfig & SpEL

Computación en Internet 2 — Semana 3

Facultad de Ingeniería — Universidad Icesi

Universidad ICESI

XML Monolítico

Verbosidad

Typo Errors

Fricción del XML Tradicional

En las primeras versiones de Spring (1.x/2.x), la configuración residía enteramente en archivos XML (`applicationContext.xml`).

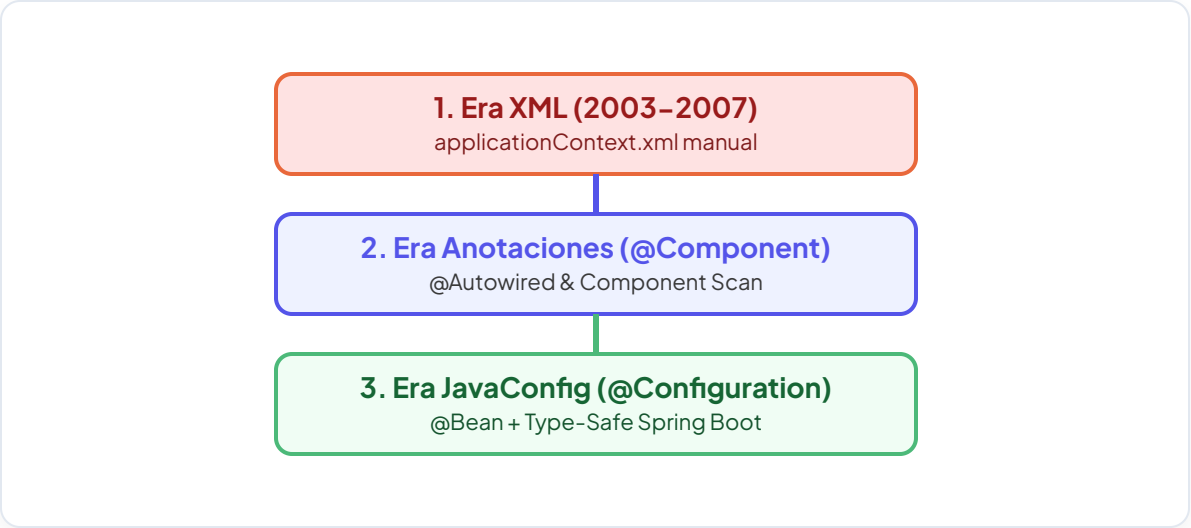
- **Verbosidad Extrema:** Cientos de líneas XML para declarar cada bean y sus argumentos.
- **Sin Chequeo de Compilación:** Errores tipográficos en nombres de clases solo fallaban al desplegar en producción.
- **Desconexión con el Código:** Modificar una clase Java exigía actualizar manualmente el XML.

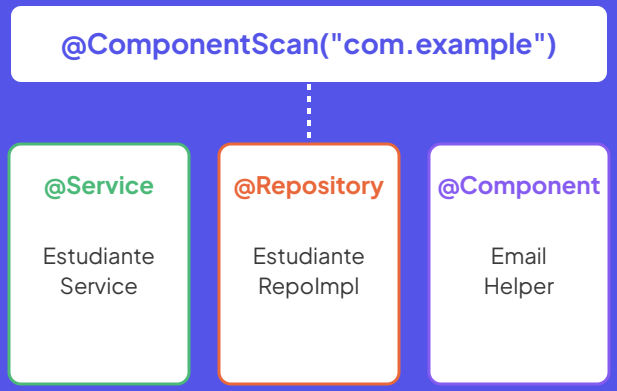
Evolución de la Configuración en Spring

De XML a JavaConfig

Spring ha evolucionado para reducir el código repetitivo (*boilerplate*) manteniendo el desacoplamiento:

- ✓ **Spring 1.x / 2.0:** Configuración XML pura y explícita.
- ✓ **Spring 2.5 / 3.x:** Introducción de Anotaciones y Component Scan.
- ✓ **Spring 4.x / 6.x+:** JavaConfig seguro y Spring Boot autodeterminadamente declarativo.





Component Scanning: Descubrimiento Automático

Con `@ComponentScan`, Spring inspecciona dinámicamente los paquetes Java registrando automáticamente las clases anotadas como Beans.

- ✓ El paquete base (ej. `com.example`) es escaneado recursivamente.
- ✓ Instancia y registra Beans en el Contenedor IoC sin código XML.

Anotaciones Estereotipo de Spring

@COMPONENT

Marcador genérico para cualquier componente Java administrado por el contenedor de Spring (helpers, utilidades).

@REPOSITORY

Capa de Acceso a Datos (DAO). **Traduce automáticamente excepciones de BD** a la jerarquía `DataAccessException`.

@SERVICE

Capa de Lógica de Negocio. Almacena reglas empresariales, validaciones y orquestación de transacciones.

@CONTROLLER / @RESTCONTROLLER

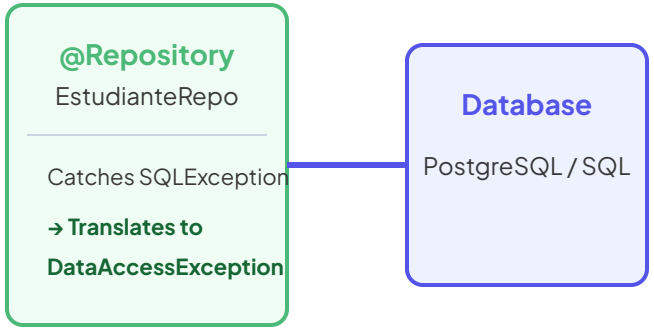
Capa de Presentación Web / API REST. Procesa peticiones HTTP, parsea JSON y retorna respuestas.

Capa de Persistencia (@Repository)

Traducción de Excepciones

Al anotar la clase con `@Repository`, Spring intercepta errores de base de datos nativos (SQL, JPA) y los convierte a excepciones unificadas de Spring.

```java
package com.example.repository;

import org.springframework.stereotype.Repository;

@Repository
public class EstudianteRepositoryImpl
  implements EstudianteRepository {

  public String findNombreEstudiante() {
    return "Juan Pérez";
  }
}
```



Capa de Negocio e Inyección por Constructor

Buenas Prácticas de Inyección

Usar inyección por constructor declarando campos `final` garantiza inmutabilidad y facilita unit testing sin levantar Spring.

> **Nota:** En Spring 4.3+, si hay un solo constructor, `@Autowired` es opcional.

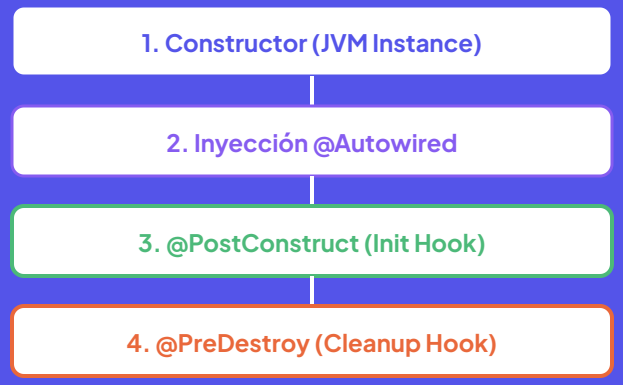
```java
@Service
public class EstudianteServiceImpl implements EstudianteService {

  private final EstudianteRepository repository;

  @Autowired // Inyección por constructor (Recomendada)
  public EstudianteServiceImpl(EstudianteRepository repository) {
    this.repository = repository;
  }

  public String obtenerDetalle() {
    return repository.findNombreEstudiante();
  }
}
```

Alcances (@Scope): Singleton vs Prototype

Comportamiento en Memoria

Por defecto, todo bean marcado con `@Component` es **Singleton** (una única instancia compartida por el contenedor).

```
@Component
@Scope("prototype") // Instancia nueva por petición
public class CarritoCompras { ... }
```



Ciclo de Vida: @PostConstruct y @PreDestroy

Reemplazando los atributos XML `init-method` y `destroy-method` con anotaciones estándar de Jakarta:

- ✓ **@PostConstruct:** Se ejecuta tras instanciar e inyectar todas las dependencias.
- ✓ **@PreDestroy:** Se ejecuta antes de destruir el Bean al cerrar el contexto.

Hooks de Inicialización y Limpieza

Casos de Uso

@PostConstruct: Precargar cachés de BD, validar URLs o abrir conexiones a servicios.

@PreDestroy: Cerrar sockets, liberar hilos de ejecución o vaciar buffers en disco.

```
@Service
public class ConexionService {

  public ConexionService() {
    System.out.println("1. Constructor JVM");
  }

  @PostConstruct
  public void init() {
    System.out.println("2. @PostConstruct: Recursos listos.");
  }

  @PreDestroy
  public void cleanup() {
    System.out.println("3. @PreDestroy: Liberando conexiones...");
  }
}
```

Java Configuration (@Configuration y @Bean)

¿Por qué JavaConfig?

Cuando requerimos instanciar clases de **librerías de terceros** (que no podemos anotar con `@Component`), usamos clases `@Configuration` con métodos `@Bean`.

```java
@Configuration
@ComponentScan(basePackages = "com.example")
public class AppConfig {

  @Bean
  public String nombreAplicacion() {
    return "Sistema DocuKelo";
  }

  @Bean(initMethod = "init", destroyMethod = "cleanup")
  @Scope("singleton")
  public ExternalLibraryService externalService() {
    return new ExternalLibraryService();
  }
}
```

Universidad ICESI

AppConfig.class

AnnotationConfig

Zero XML

AnnotationConfigApplicationContext

Al eliminar el XML por completo, instanciamos el contenedor IoC pasando la clase `@Configuration`:

```
AnnotationConfigApplicationContext context =
  new AnnotationConfigApplicationContext(AppConfig.class);

EstudianteService service =
  context.getBean(EstudianteService.class);
```

Universidad ICESI

Desambiguación e Inyección de Propiedades

Resolución de conflictos con @Qualifier / @Primary y configuración externa

Universidad ICESI

Dos Beans

NoUniqueBean Exception

Ambigüedad: NoUniqueBeanDefinitionException

¿Qué ocurre cuando solicitamos inyectar una interfaz con `@Autowired` y existen **2 o más clases** que la implementan?

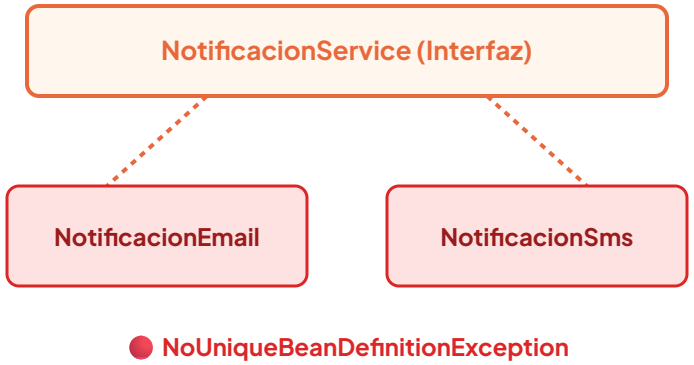
- Spring no sabe cuál elegir e interrumpe el arranque de la aplicación.
- Lanza la excepción `NoUniqueBeanDefinitionException`.

Escenario de Conflicto de Dependencias

Dos Implementaciones

Interfaz `NotificacionService` implementada por `NotificacionEmailServiceImpl` y `NotificacionSmsServiceImpl`.

```
// Inyección ambigua:
@Autowired
public UsuarioController(NotificacionService service) {
 // 🔴 ERROR: ¿Email o SMS?
}
```



Selección Explícita con @Qualifier

@Qualifier("nombreBean")

Especifica el identificador exacto del Bean (por defecto el nombre de la clase en *camelCase*).

```
@Service
public class UsuarioController {

  private final NotificacionService service;

  @Autowired
  public UsuarioController(
    @Qualifier("notificacionSmsServiceImpl")
    NotificacionService service) {

    // 🟢 Inyecta explícitamente la versión SMS
    this.service = service;
  }
}
```

Opción Preferida por Defecto con @Primary

@Primary

Marca una clase como la opción prioritaria cuando existe ambigüedad, evitando usar `@Qualifier` en cada punto de inyección.

Regla de Precedencia:

`@Qualifier` **SIEMPRE** tiene mayor prioridad y sobrescribe a `@Primary`.

```
@Service
@Primary // 🟢 Opción por defecto
public class NotificacionEmailServiceImpl
  implements NotificacionService {

  public void enviarMensaje(String msg) {
    System.out.println("EMAIL Prioritario: " + msg);
  }
}
```

Universidad ICESI

application.properties

@PropertySource

Desacoplamiento

Configuración Externa con @PropertySource

Desacopla credenciales, URLs de servicios y parámetros de entorno en archivos `application.properties`.

```
@Configuration
@PropertySource("classpath:application.properties")
public class AppConfig { ... }
```

Inyección de Propiedades con @Value

Sintaxis ${...}

Extrae valores del archivo `.properties` y realiza conversión automática a tipos primitivos (int, boolean).

Soporta valores por defecto si la propiedad no existe: `${app.timeout:3000}`.

```java
@Service
public class SistemaConfigService {

  @Value("${app.nombre}")
  private String nombreApp;

  @Value("${app.max-usuarios}")
  private int maxUsuarios;

  @Value("${app.timeout:3000}") // Default 3000
  private int timeout;
}
```

Resumen de Inyección Avanzada

@QUALIFIER

Resuelve ambigüedad especificando el nombre exacto del bean a inyectar.

@PRIMARY

Define la implementación preferida por defecto sin modificar el punto de inyección.

@PROPERTYSOURCE

Carga el archivo `application.properties` desde el classpath en la clase `@Configuration`.

@VALUE("${...}")

Extrae y convierte valores de propiedades a variables Java automáticamente.

Universidad ICESI

Spring Expression Language (SpEL)

Evaluación dinámica de expresiones en tiempo de ejecución con #{...}

Universidad ICESI

Property Placeholder ${...}

Extrae texto literal de .properties

SpEL Engine #{...}

Ejecuta lógica, Math, UUID, Beans

Evaluador Dinámico SpEL

SpEL permite evaluar expresiones complejas, cálculos matemáticos e invocar métodos al instanciar Beans.

- ✓ Sintaxis `#{expresion}` evaluada en runtime.
- ✓ Puede consultar otros beans del ApplicationContext.

Property Placeholder ${} vs SpEL #{}

${...} Property Placeholder

Lectura pasiva de llaves en `application.properties`. No realiza cálculos ni lógica.

```
@Value("${app.nombre}")
private String nombre;
```

#{...} SpEL Expression

Motor activo de evaluación en tiempo de ejecución. Realiza cálculos, llama métodos y evalúa booleanos.

```
@Value("#{10 * 5 + 20}")
private int total; // Asigna 70
```

Operaciones Matemáticas y Lógicas

Evaluaciones Directas

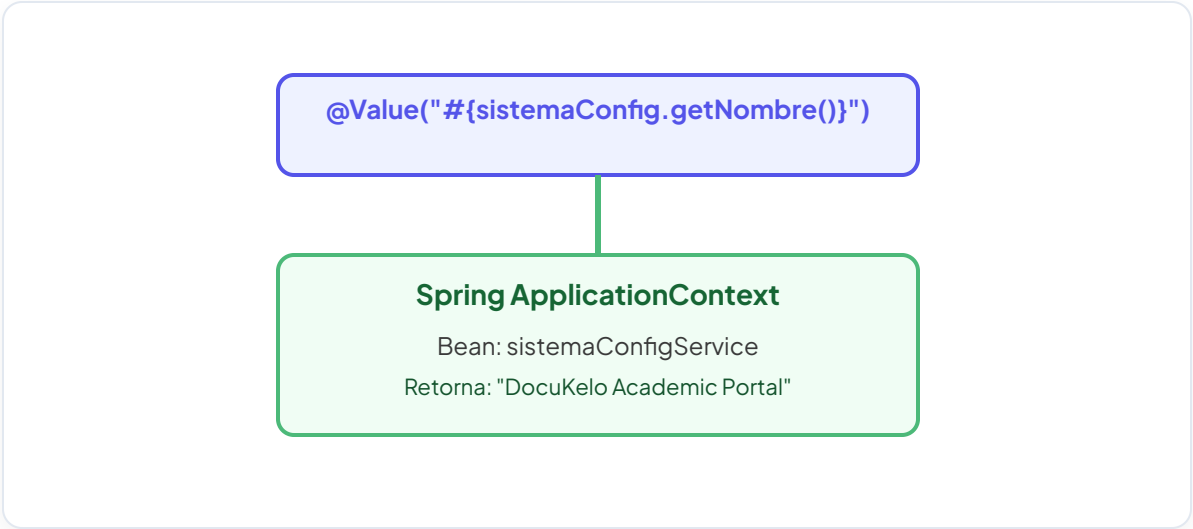
SpEL soporta aritmética básica, comparadores relacionales (`>` , `<` , `==`) y operadores lógicos (`and` , `or`).

```
@Service
public class CalculoSpELService {

  @Value("#{10 * 5 + 20}")
  private int calculoMatematico; // 70

  @Value("#{sistemaConfigService.maxUsuarios > 100}")
  private boolean esGranEscala; // true/false

  @Value("#{'DocuKelo'.toUpperCase() + ' v2.5'}")
  private String tituloFormateado;
}
```

Acceso a Métodos y Propiedades de Beans

Consulta al ApplicationContext

SpEL puede referenciar cualquier bean registrado por su nombre e invocar sus métodos directamente: `#{nombreBean.metodo()}`.



Clases Estáticas con Sintaxis T(...)

Operador T()

Para invocar métodos estáticos o constantes de clases nativas de Java (ej. `java.lang.Math`, `java.util.UUID`), envolvemos el nombre de la clase en `T(...)`.

```
@Service
public class ProcesadorService {

  @Value("#{T(java.lang.Math).PI}")
  private double valorPi;

  @Value("#{T(java.lang.Math).random() * 100}")
  private double aleatorio;

  @Value("#{T(java.util.UUID).randomUUID().toString()}")
  private String uuidSesion;
}
```

Operador Ternario y Expresiones Anidadas

Combinando ${} y #{}

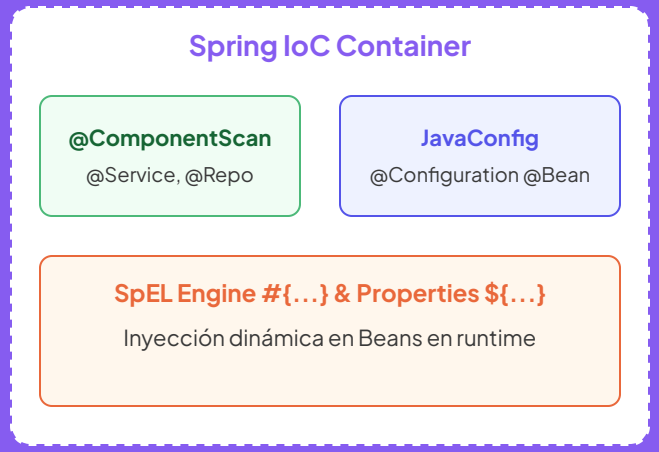
Podemos anidar una propiedad leída de `.properties` dentro de una expresión SpEL para operarla dinámicamente.

```
// Operador Ternario SpEL
@Value("#{estudiante.promedio >= 3.0 ? 'Aprobado' : 'Reprobado'}")
private String estadoEstudiante;

// Combinando property ${} dentro de SpEL #{...}
@Value("#{${app.max-usuarios} * 2}")
private int limiteDuplicado;
```

Sintaxis y Operadores Fundamentales de SpEL

Categoría SpEL	Ejemplo de Expresión	Resultado / Descripción
Aritmético	`#{100 / 4 + 5}`	Evalúa y asigna `30`.
Relacional	`#{50 > 20 and 10 == 10}`	Evalúa condición booleana (`true`).
Invocación de Bean	`#{estudianteService.getNombre()}`	Llama método de un bean registrado.
Clase Estática T()	`#{T(java.lang.Math).abs(-45)}`	Retorna constante o método estático (`45`).
Ternario Condicional	`#{user.active ? 'OK' : 'LOCKED'}`	Retorna valor según condición.

Universidad ICESI



Arquitectura Integral del Contenedor IoC

Visión holística: Beans, ApplicationContext, ComponentScan, JavaConfig, Properties y SpEL Engine.

Ejemplo Integrador: AuditoriaComponent

Reuniendo Todo

Combinación de `@Qualifier` (desambiguación), `@Value` (propiedades) y `SpEL` (generación de UUID dinámico).

```
@Component
public class AuditoriaComponent {

  private final NotificacionService service;

  @Value("${app.nombre:Portal}")
  private String nombreApp;

  @Value("#{'SESS-' + T(java.util.UUID).randomUUID().toString().substring(0,8)}")
  private String sessionId;

  @Autowired
  public AuditoriaComponent(
    @Qualifier("notificacionEmailServiceImpl") NotificacionService s) {
    this.service = s;
  }
}
```

Universidad ICESI

Cuadro Comparativo: XML vs Anotaciones vs JavaConfig

Criterio	XML (Legacy)	Anotaciones (@Component)	JavaConfig (@Bean)
Ubicación	Archivos `.xml` externos	En las clases Java propias	Clases `@Configuration`
Detección	Manual por cada bean	Automática con `@ComponentScan`	Métodos anotados con `@Bean`
Type-Safety	Baja (Errores en runtime)	Alta (Chequeo en compilación)	Máxima (Código Java puro)
Caso de Uso	Mantenimiento antiguo	Componentes del proyecto propio	Librerías de terceros / Complejos

Errores Comunes en Spring Core

OLVIDAR @COMPONENTSCAN

Spring no descubre los beans si no están dentro del paquete base especificado en la configuración.

CONFUNDIR ${} Y #{}

Usar `${}` para intentar cálculos matemáticos o llamadas a métodos en lugar de utilizar `#{}`.

FALTA DE @PROPERTYSOURCE

`@Value("${...}")` inyecta la clave literal si no se registró el archivo `.properties` en `@Configuration`.

DEPENDENCIAS CIRCULARES

Bean A inyecta a B y B inyecta a A por constructor, generando un bucle infinito en el arranque.

Checklist de Aprendizaje: Semana 3

1. ESTEREOTIPOS

Dominio de `@Component`, `@Service`, `@Repository` (con traducción de excepciones) y `@Controller`.

2. CICLO DE VIDA

Manejo de `@PostConstruct`, `@PreDestroy`, y diferencia entre scopes `singleton` y `prototype`.

3. DESAMBIGUACIÓN

Resolución de conflictos de dependencias múltiples usando `@Qualifier` y `@Primary`.

4. SPEL & PROPERTIES

Configuración externa con `@Value("${...}")` y evaluación de expresiones dinámicas con `#{...}`.

Universidad ICESI

Spring Core Dominado

Preparados para Spring Boot y desarrollo de Servicios Web REST