Universidad ICESI

Spring Framework

Inversión de Control, Contenedor de Beans y Arquitectura Web en Java

Universidad ICESI

Principios de Diseño, Patrones e IoC

Principios vs. Patrones vs. Frameworks

Jerarquía de Abstracción

Diferencia entre filosofía de diseño, estructura del código e infraestructura ejecutable:

Principios de Diseño:
Reglas de alto nivel (SOLID). Indican QUÉ cualidad buscar sin imponer código.

Patrones de Diseño:
Soluciones probadas (Factory, Strategy). Indican CÓMO estructurar clases.

Framework (Spring):
Herramienta ejecutable que automatiza los patrones mediante contenedores IoC.

Los 5 Principios SOLID de Diseño

Single Responsibility & Open/Closed

S – Single Responsibility (SRP):
Una clase debe tener una sola razón para cambiar; solo cumple una función específica.

O – Open/Closed (OCP):
Abierto para extensión pero cerrado para modificación mediante abstracciones.

L – Liskov Substitution (LSP):
Subclases deben poder reemplazar a sus clases base sin alterar el programa.

Interface Segregation & Dependency Inversion

I – Interface Segregation (ISP):
Preferir varias interfaces específicas en lugar de una interfaz sobrecargada.

D – Dependency Inversion (DIP):
Módulos de alto nivel no dependen de bajo nivel; ambos dependen de abstracciones.

Inversión de Dependencias (DIP) en Práctica

Sin DIP (Alto Acoplamiento)

La clase de negocio depende directamente de la implementación concreta:

```
public class Servicio {
  // Acoplamiento fuerte a la clase concreta
  private EmailSender sender = new EmailSender();
}
```

Imposible probar de forma aislada o cambiar el canal de mensajería sin recompilar.

Con DIP & Spring IoC (Bajo Acoplamiento)

La clase depende de una interfaz abstracta inyectada por Spring:

```
public class Servicio {
  // Depende de la interfaz MessageSender
  private MessageSender sender;

  public Servicio(MessageSender sender) {
    this.sender = sender;
  }
}
```

Desacoplamiento total. Permite cambiar implementaciones en metadatos XML.

Universidad ICESI

Control Delegado

POJOs Desacoplados

Testabilidad Total

Inversion of Control (IoC) & Inyección

El Cambio de Paradigma IoC

En la programación tradicional, el desarrollador controla el flujo instanciando objetos con `new`.

Control Delegado:
La creación, gestión y ensamblado de objetos se delega al **Contenedor IoC de Spring.**

- **Programador:** Declara reglas en metadatos (XML o Anotaciones).
- **Framework:** Instancia las clases POJO e inyecta dependencias en tiempo de ejecución.

Universidad ICESI

02. El Ecosistema y Módulos de Spring Framework

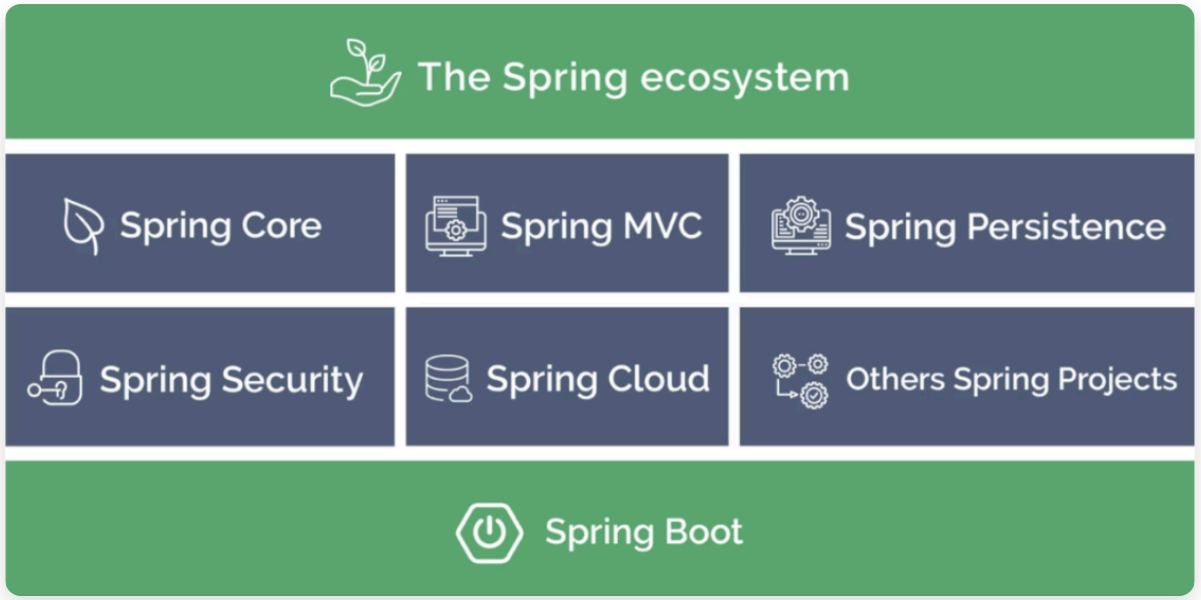
Arquitectura Modular del Ecosistema Spring

Eliminando el "Plumbing Code"

Spring no es un monolito, sino una familia modular de proyectos diseñados para eliminar el código repetitivo de infraestructura.

Foco en el Negocio:
Desarrolladores escriben reglas del dominio sin preocuparse por la fontanería de conexiones.

Integración Seleccionable:
Cada módulo (Core, Data, Web, Security) se agrega según las necesidades de la app.



Universidad ICESI

Módulos Clave del Ecosistema Spring

SPRING CORE

El motor fundamental. Proporciona el Contenedor IoC, BeanFactory, ApplicationContext e Inyección de Dependencias.

SPRING DATA / PERSISTENCE

Abstracción de acceso a datos. Simplifica JDBC, transacciones declarativas y soporte ORM con JPA/Hibernate.

SPRING WEB (MVC)

Arquitectura de presentación basada en Servlets. Maneja peticiones HTTP mediante DispatcherServlet y Controllers.

SPRING BOOT & SECURITY

Auto-configuración de aplicaciones empresariales con servidores embebidos (Tomcat) y seguridad robusta.

Arquitectura en Capas: Model–Service–Repository

Matriz de Responsabilidades

Capa	Componente	Responsabilidad
Presentación	Servlet / Controller	Atiende HTTP y responde vistas/JSON.
Servicio	Service POJO	Lógica y reglas del negocio.
Repositorio	DAO / Repository	Acceso y consultas a base de datos.
Modelo	Domain Entity	POJOs transportados entre capas.

Reglas de Comunicación

- **Paso 1:** El Servlet invoca a la Capa de Servicio exclusivamente mediante su **Interfaz Java**.
- **Paso 2:** La Capa de Servicio procesa validaciones y consulta la Capa de Repositorio mediante interfaces.
- **Paso 3:** El Repositorio lee la base de datos y retorna objetos del **Modelo**.

Aislamiento Estricto:

Ninguna capa conoce los detalles de implementación de la capa inferior.

Universidad ICESI

03. El Contenedor IoC y Configuración XML

El Contenedor IoC & Spring Beans

¿Qué es un Spring Bean?

Un **Spring Bean** es cualquier objeto Java cuya instanciación, ensamblado y ciclo de vida son gestionados por el contenedor IoC dentro de la JVM.

POJOs Limpios:
Sin herencia de clases del framework ni interfaces propietarias.

Ensamblado Declarativo:
Las relaciones se definen en el archivo XML (`applicationContext.xml`).

Tipos de Contenedores IoC

BeanFactory:
Contenedor básico con soporte de DI y *lazy loading* (instancia solo al solicitar).

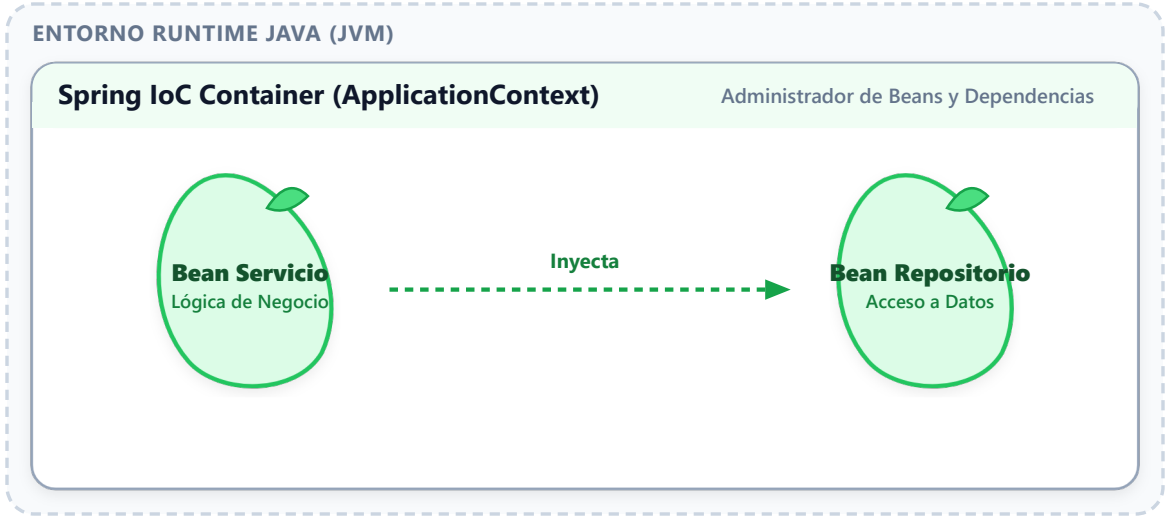
ApplicationContext (Recomendado):
Extiende BeanFactory con *eager loading*, internacionalización, eventos e integración web.

Arquitectura Interna del Contenedor IoC

Proceso de Carga y Ensamblado

El contenedor IoC lee la configuración y construye en memoria el grafo de objetos vivos (beans):

- **Lectura:** Carga de metadatos XML desde applicationContext.xml.
- **Instanciación:** Creación de los objetos POJO en la memoria Heap.
- **Inyección:** Vinculación de dependencias entre Repositorio y Servicio.



Declaración de Beans e Inyección en XML

Archivo applicationContext.xml

Sintaxis declarativa limpia para definir e inyectar beans sin usar anotaciones:

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://www.springframework.org/schema/beans
      http://www.springframework.org/schema/beans/spring-beans.xsd">

      <!-- 1. Bean de Repositorio -->
      <bean id="estudianteRepositoryBean"
     class="com.example.repository.EstudianteRepositoryInMemory" />

      <!-- 2. Bean de Servicio con Inyección por Constructor -->
      <bean id="estudianteServiceBean"
     class="com.example.service.EstudianteServiceImpl">
            <constructor-arg ref="estudianteRepositoryBean" />
      </bean>

</beans>
```

04. Alcances (Scopes), Ciclo de Vida y Ejecución Main

Alcances (Scopes): Singleton vs. Prototype

Singleton (Scope por Defecto)

Una **única instancia** compartida en todo el `ApplicationContext`.

- Ideal para componentes sin estado (*Stateless*): Servicios y Repositorios.
- Reutiliza la misma referencia en memoria optimizando la RAM.

Prototype (Scope por Demanda)

Una **nueva instancia** creada cada vez que se solicita el bean.

- Requerido para objetos con estado (*Stateful*): Carritos de compra, buffers de reportes.
- Garantiza inmunidad a condiciones de carrera en entornos multihilo.

Universidad ICESI

1. Instanciación

2. Inyección DI

3. Init Method

4. Bean Listo

5. Destroy Method

Ciclo de Vida del Bean en Spring

Fases de Gestión de Memoria

El contenedor IoC administra el ciclo de vida del bean desde su creación hasta su eliminación final:

1. **Instanciación:** Invocación del constructor Java.
2. **Inyección DI:** Asignación de dependencias (`ref`).
3. **Inicialización:** Ejecución de `init-method="..."` .
4. **Bean Listo:** Servidor listo para responder peticiones.
5. **Destrucción:** Ejecución de `destroy-method="..."` al apagar.

Universidad ICESI

Carga del Contenedor en Aplicaciones Main

Bootstrapping con ClassPathXmlApplicationContext

```
package com.example;

import org.springframework.context.ApplicationContext;
import org.springframework.context.support.ClassPathXmlApplicationContext;
import com.example.service.EstudianteService;

public class MainApp {
    public static void main(String[] args) {
        // 1. Inicializar el contenedor IoC desde el archivo XML
        ApplicationContext context =
                new ClassPathXmlApplicationContext("applicationContext.xml");

        // 2. Obtener el bean de servicio inyectado
        EstudianteService servicio =
                (EstudianteService) context.getBean("estudianteServiceBean");

        // 3. Ejecutar la lógica de negocio
        servicio.listarEstudiantes().forEach(System.out::println);
    }
}
```

Universidad ICESI

05. Integración: Servlets Tomcat & Spring IoC

Integración Web: El Anti-Patrón en Servlets

El Anti-Patrón (Error Común)

```
@WebServlet("/estudiantes")
public class EstudianteServlet extends HttpServlet {
  protected void doGet(HttpServletRequest req, ...) {
    // Recrea el XML en CADA petición HTTP
    ApplicationContext ctx =
      new ClassPathXmlApplicationContext("appContext.xml");
  }
}
```

Consecuencias de Recrear el XML

- **Destrucción del Rendimiento:** Lee y parsea el archivo XML desde disco en cada clic HTTP.
- **Pérdida del Scope Singleton:** Se crean miles de contenedores e instancias huérfanas en memoria.
- **Fugas de Memoria (Memory Leaks):** El Garbage Collector colapsa intentando liberar contextos.

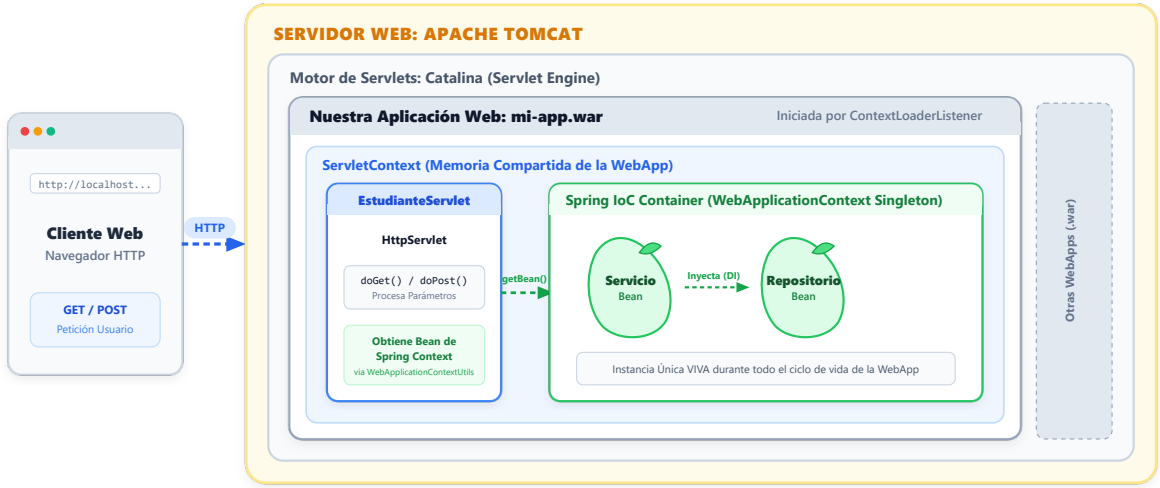
Tomcat Catalina & ContextLoaderListener

Carga Única al Arrancar Tomcat

Para integrar Spring con Servlets de forma profesional, el `ApplicationContext` debe instanciarse **una sola vez al arrancar Tomcat**.

ContextLoaderListener:

Listener web de Spring que escucha el inicio de la WebApp y guarda el contenedor dentro del `ServletContext` de Tomcat.



Configuración Estándar en web.xml

Descriptor de Despliegue Web

```
<web-app xmlns="https://jakarta.ee/xml/ns/jakartaee" version="6.0">

    <!-- 1. Ubicación del archivo de contexto XML -->
    <context-param>
        <param-name>contextConfigLocation</param-name>
        <param-value>classpath:applicationContext.xml</param-value>
    </context-param>

    <!-- 2. Listener que inicializa el ApplicationContext al arrancar Tomcat -->
    <listener>
        <listener-class>
            org.springframework.web.context.ContextLoaderListener
        </listener-class>
    </listener>

</web-app>
```

Consumo Correcto de Beans en un Servlet

Recuperación en el método init()

```
@WebServlet("/estudiantes")
public class EstudianteServlet extends HttpServlet {

  private EstudianteService servicio;

  @Override
  public void init() throws ServletException {
    // Recuperar ApplicationContext desde ServletContext
    ApplicationContext context =
      WebApplicationContextUtils
        .getRequiredWebApplicationContext(
          getServletContext()
        );

    this.servicio = (EstudianteService)
      context.getBean("estudianteServiceBean");
  }
}
```

Ventajas del Enfoque Correcto

- **Rendimiento Óptimo:** El XML se lee una sola vez al arrancar el Servlet.
- **Reuso de Instancias:** Todas las peticiones HTTP comparten la misma instancia del servicio.
- **Zero Memory Leaks:** Integración limpia con el ciclo de vida de Tomcat.

Flujo Completo: De la Petición HTTP al Bean

1. PETICIÓN CLIENTE

El usuario realiza una solicitud HTTP GET desde el navegador hacia Tomcat.

2. CATALINA ENGINE

Tomcat intercepta la petición y la enruta al `EstudianteServlet`.

3. INVOCACIÓN BEAN

El Servlet invoca al bean `EstudianteService` recuperado en `init()`.

4. RESPUESTA HTTP

El Servlet procesa el resultado del servicio y responde JSON/HTML al navegador.