


# Spring Framework

Contexto: **Java Enterprise (JEE)**

Formador: **Ezequiel Llarena Borges**

## conocimientos necesarios

- Core Java SE
- Programación Orientada a Objetos
- Maven (pom.xml)
- POJO's
- Java Bean vs EJB
- Principios SOLID
- Patrones de diseño



Gestión de librerías, no necesario para Spring

## **1. Cultura General**

## **2. Conceptos y Principios básicos**

- Principio de Inversión de Dependencia (DIP)
- Inversión de Control (IoC)
- Inyección de Dependencias (DI)

## **3. Spring Core Container**

## **4. Módulos**

## **5. Ventajas**

## **6. Bibliografía**



## cultura general...

- Rod Johnson
- “Expert One-to-One J2EE Design and Development” 2002
- Aplicaciones Empresariales Java EE
- Experiencia y buenas prácticas
- Spring 1.0 Marzo 2004
- Spring 3.0 Diciembre 2009
- Spring 4.0 Enero 2013
- Spring 5.0.2 Latest Release Octubre 2017

# ¿Qué es Spring Framework?

Plataforma Java *open source* para el desarrollo de aplicaciones empresariales Java

Elementos básicos:

- Servicios *enterprise*
- Estereotipos configurables
- Inyección de dependencias

## conceptos básicos

- Acoplamiento
- Cohesión
- Contenedor de Inversión de Control (IoC) usando Java Reflexión
- Inyección de dependencias



### **API Reflection**

Permite manipular  
nuestras clases en  
tiempo de ejecución

# Principio de Inversión de Dependencia (DIP)

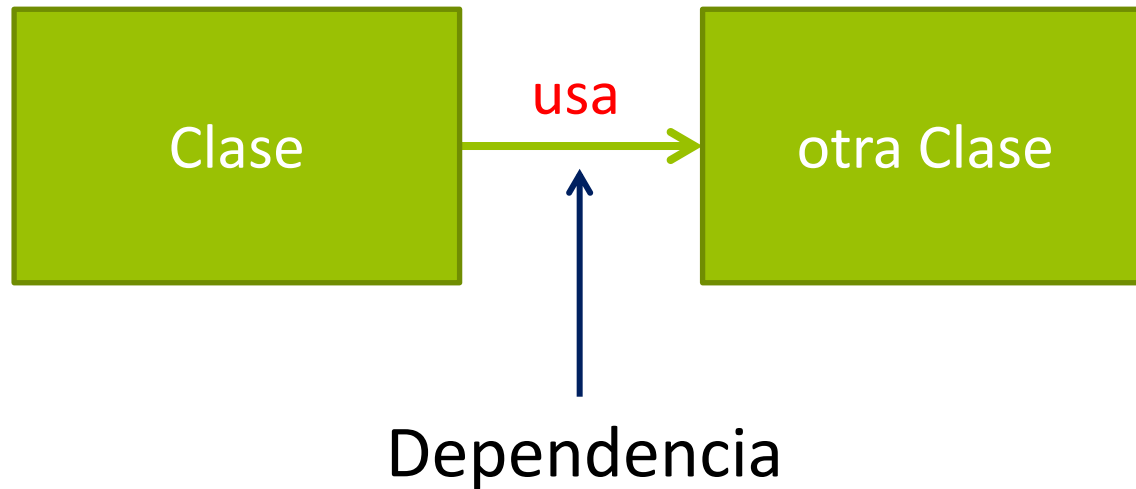
- Los módulos de alto nivel no deben depender de los módulos de bajo nivel
- Abstracciones sobre implementaciones
- Nos ayudará a crear código **desacoplado**
- El patrón **IoC** es una aplicación de este principio

# Inversión de Control (IoC)

- Implementación del DIP (*Dependency Inversion Principle*)
- Principio de Hollywood (“No nos llames, nosotros te llamamos”)
- Estilo de programación donde un agente externo controla el flujo de la aplicación
- Término genérico que se implementa de distintas maneras:
  - ✓ Service Locator (otro patrón de diseño de software)
  - ✓ Events
  - ✓ Delegates
  - ✓ Inyección de dependencias (DI)




# Inyección de Dependencias (DI)



# Dependencia

```
public class Customer {  
    public Logger log;  
    public Customer() {  
        log = new Logger();  
    }  
}
```

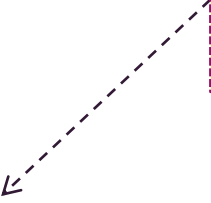
instancias de  
Customer  
FUERTEMENTE  
ACOPLADAS a la  
clase Logger



# Desacoplar

```
public class Customer {  
    public Logger log;  
    public Customer(Logger obj) {  
        log = obj;  
    }  
}
```

Inyección de  
Dependencias en el  
Constructor



# Inyección de Dependencias (DI)

- Patrón de Diseño de Software
- Inyección de dependencia con Reflection
- Subtipo de IoC
- Objetivo: código fácil de mantener
- Provee a los objetos lo que el objeto necesita:
  - ✓ Constructor
  - ✓ Propiedad (set)
  - ✓ Servicio / interfaz

# Inyección de Dependencia

- Elimina la dependencia entre dos clases
- Crea automáticamente instancias de una clase

# Spring Core Container

- Es un IoC Container
- Gestiona ciclo de vida de los objetos (beans)
  - ✓ Crearlos
  - ✓ Enlazarlos
  - ✓ Configurarlos
  - ✓ Destruirlos

# Core Container

- Contenedor de Spring
- Fábrica de Beans de Spring
- ApplicationContext
- Objeto basado en patrón de diseño Factory

## Bean

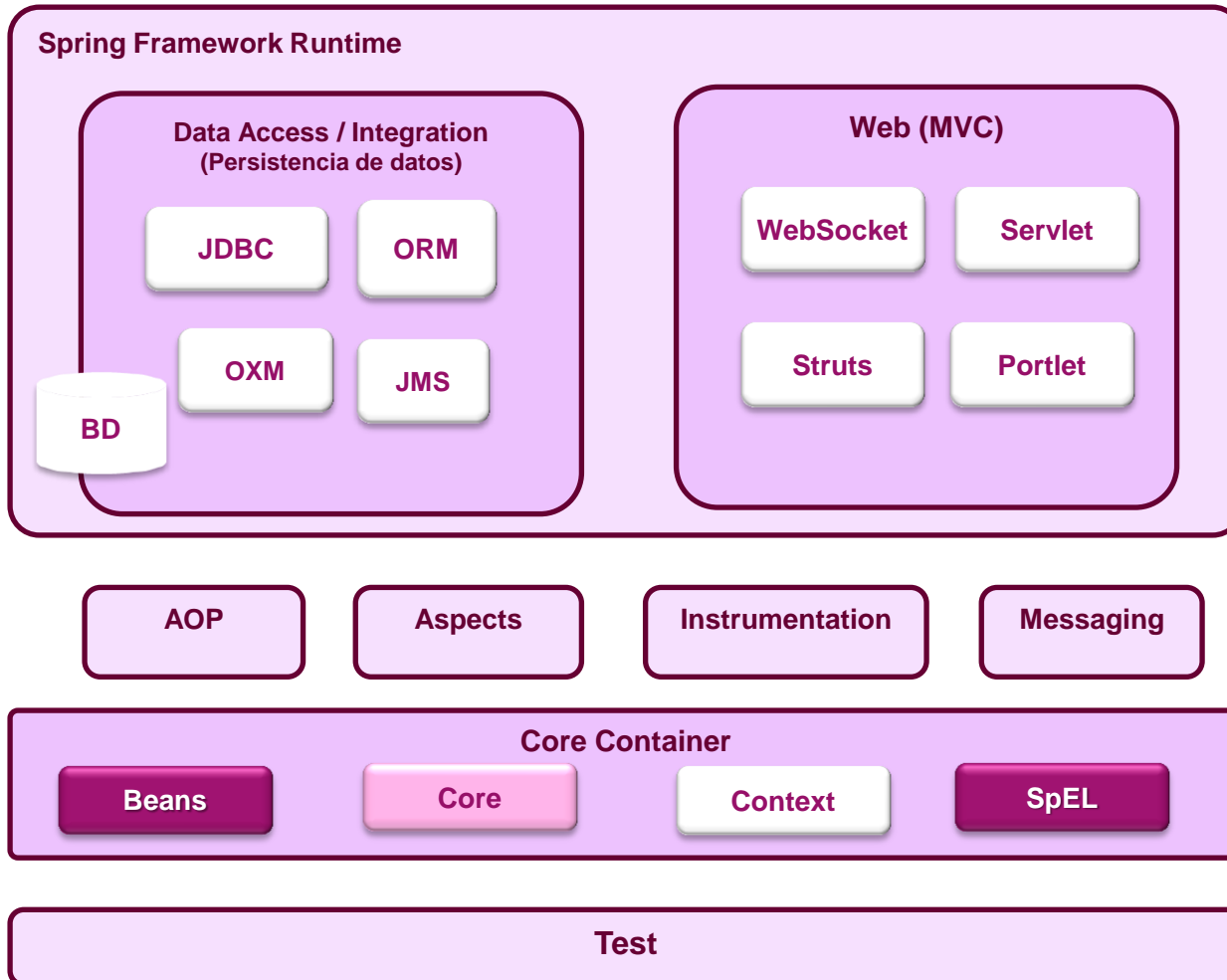
- Los beans son la manera que tiene de denominar Spring a los **objetos Java de los que se encarga**, es decir aquellos que **se encuentren en el contenedor de Spring**.



# Otros IoC Container

- Java:
  - ✓ PicoContainer
  - ✓ NanoContainer
  - ✓ Peapod
  - ✓ Gravity
  - ✓ Google Guice
- .Net:
  - ✓ Unity
  - ✓ Autofac
  - ✓ Ninject
  - ✓ Structure Map

# Módulos



# Ventajas de Spring

- Simplifica aplicaciones J2EE/JEE (POJO's, Java Beans)
- Framework Ligero + No Intrusivo
- Flexibilidad (Integración con otras herramientas)
- Reduce código repetitivo (p.e. JDBC)
- Alta cohesión
- Uso de Anotaciones
- Inyección de Dependencias (patrón de diseño de software)
- Bajo Acoplamiento (DI + Programación orientado a Interfaces)
- Ahorrar Tiempo + Coste en el desarrollo
- Testing

# Críticas de Spring

- XML verboso
- Container no es ligero

# Recursos



Documentación Oficial del Proyecto

<https://spring.io/docs>



# Formas de Configuración de Beans

- XML
- @notaciones
- JavaConfig

## ID XML en el Constructor

```
<bean id="bean_id" class="com.example.SimpleSPImpl">  
    <constructor-arg name="ds" ref="datasource" />  
    <constructor-arg name="spName" value="spNameTest" />  
</bean>
```

```
<!-- Definición del datasource de ejemplo -->  
<bean id="datasource" class="com.mchange.ComboPooledDataSource">  
    ...  
</bean>
```

## c-namespace

```
<bean id="bean_id" class="com.example.SimpleSPImpl">  
    <constructor-arg name="ds" ref="datasource" />  
    <constructor-arg name="spName" value="spNameTest" />  
</bean>
```

```
<bean id="bean_id" class="com.example.SimpleSPImpl"  
    c:ds-ref="datasource"  
    c:spName="spNameTest" />
```

```
<!-- Definición del datasource de ejemplo -->  
<bean id="datasource" class="com.mchange.ComboPooledDataSource">  
    ...  
</bean>
```



## ID XML en el Setter

```
<bean id ="miGestorUsuarios"  
      class="es.ua.jtech.spring.negocio.GestorUsuarios">  
<!-- Cuidado, la propiedad debe llamarse igual que en el fuente Java -->  
      <property name="udao" ref="miUsuariosDAO"/>  
</bean>
```

## p-namespace

```
<!-- Service Beans -->
<bean id="bookService" class="com.bookstore.services.BookService"
    p:bookId="book1"
    p:bookDao-ref="bookDao" />

<bean id="purchasingService"
class="com.bookstore.services.PurchasingService"
    p:bookServiceInterface-ref="bookService"
    p:accountServiceInterface-ref="accountService"></bean>

<bean id="accountService" class="com.bookstore.services.AccountService"
    p:accountDao-ref="accountDao" />
```

# namespaces p y c en STS

## Namespaces

### Configure Namespaces

Select XSD namespaces to use in the configuration file

- ☐ aop - <http://www.springframework.org/schema/aop>
- ☒ beans - <http://www.springframework.org/schema/beans>
- ☒ c - <http://www.springframework.org/schema/c> 
- ☐ cache - <http://www.springframework.org/schema/cache>
- ☐ context - <http://www.springframework.org/schema/context>
- ☐ jdbc - <http://www.springframework.org/schema/jdbc>
- ☐ jee - <http://www.springframework.org/schema/jee>
- ☐ jms - <http://www.springframework.org/schema/jms>
- ☐ lang - <http://www.springframework.org/schema/lang>
- ☐ p - <http://www.springframework.org/schema/p>
- ☐ task - <http://www.springframework.org/schema/task>
- ☐ tx - <http://www.springframework.org/schema/tx>
- ☐ util - <http://www.springframework.org/schema/util>

# Formas de Configuración de Beans

- XML
- **@notaciones**
- JavaConfig

## Escaneo de Anotaciones

Para que Spring navegue por las clases y escanee las anotaciones:

```
<context:component-scan base-package="com.comp.beans"></context:component-scan>  
  
<!-- necesario para que busque beans en la ruta indicada por base-package -->
```

## Tipos de Beans

- Presentación
- Lógica de Negocio / Servicio
- Acceso a Datos

## Estereotipos configurables

- @Component
- @Service
- @Repository
- @Controller

## @Autowired

Permite **resolver la inyección de dependencias** de los siguiente modos:

- En el constructor de la clase
- En un atributo
- En un método setter
- En un método JavaConfig (`autowire="byName"`)



## @Autowired en el constructor

La inyección se realiza **en el momento en que el objeto es creado.**

```
@Component
public class MyController {

    private final MyBean myBean;

    @Autowired
    public MyController(MyBean myBean) {
        this.myBean = myBean;
    }
}
```

## @Autowired en el constructor (con @Value)

La inyección se realiza en el momento en que el objeto es creado.

```
@Controller
public class MyController {

    //@Value("1")
    private int id;
    private MyBean myBean;

    @Autowired
    public MyController(@Value("1") int id, MyBean myBean) {
        this.myBean = myBean;
    }
}
```

## @Autowired en el setter

Se creará el método y una vez creado, Spring inyectará el *bean* mediante dicho método.

```
@Controller
public class MyController {

    private MyBean myBean;

    @Autowired
    public void setMyBean(MyBean myBean) {
        this.myBean = myBean;
    }
}
```

## @Value en el setter

Uso de @Value para manejar parámetros en el constructor:

```
@Component
public class Direccion {

    private String calle;

    @Autowired
    public void setCalle(@Value("Calle Amparo")String calle) {
        this.calle = calle;
    }
}
```

## @Autowired sobre el atributo

Spring crea la instancia del objeto y una vez creada le inyecta la dependencia.

```
@Controller
public class MyController {

    @Autowired
    private MyBean myBean;
}
```

## Spring Bean Autowiring (XML)

```
<!--shapeBox wire by name -->
<bean id="shapeBox2" class="com.hmkcode.beans.ShapeBox" autowire="byName" />

<!--shapeBox wire by type -->
<bean id="shapeBox3" class="com.hmkcode.beans.ShapeBox" autowire="byType" />

<!--shapeBox wire by constructor -->
<bean id="shapeBox4" class="com.hmkcode.beans.ShapeBox" autowire="constructor"/>
```

spring-config.xml

## Dependency checking

```
import org.springframework.beans.factory.annotation.Autowired;
```

```
public class Customer {
```

```
    // Indicamos que no hay que satisfacer la dependencia
```

```
    @Autowired(required=false)
```

```
    private Person person;
```

```
    private int type;
```

```
    private String action;
```

```
    //getter and setter methods }}
```

## @Qualifier

```
import org.springframework.beans.factory.annotation.Autowired;  
import org.springframework.beans.factory.annotation.Qualifier;  
  
public class Customer {  
    @Autowired  
    @Qualifier("personBean1")  
    private Person person;  
    private int type;  
    private String action;  
  
    //getter and setter methods }
```



# Formas de Configuración de Beans

- XML
- @notaciones
- **JavaConfig**

# JavaConfig

```
@Configuration  
public class AppConfig {  
  
    @Bean  
    public MyService myService() {  
        return new MyServiceImpl();  
    }  
}
```

Equivalente a definir:

<bean...> en XML

## Otras anotaciones estándar JEE

- `@Inject` o `@Resource`

# Ciclo de vida de un Spring Bean

Aplicación:

- Dependencias de otros beans (pueden requerir que estén creados previamente)
- Tareas previas a la inicialización
- XML: init-method y destroy-method
- @PostConstruct y @PreDestroy
- InitializingBean y DisposableBean

## init-method

```
<!-- DAO Beans -->
```

```
<bean id="bookDao" class="com.bookstore.data.BookDao"  
      p:hibernateTemplate-ref="hibernateTemplate" />
```

```
<bean id="accountDao" class="com.bookstore.data.AccountDao"  
      init-method="createTable"<--  
      p:jdbcTemplate-ref="jdbcTemplate" />
```

Nombre de método de la  
clase AccountDao

## destroy-method

```
<!-- DataSource Beans -->
```

```
<bean id="dataSource" class="org.apache.commons.dbcp.BasicDataSource"
    destroy-method="close" ←
```

Nombre de método de la clase  
BasicDataSource

```
    p:url="jdbc:hsqldb:file:database.dat;shutdown=true"
    p:driverClassName="org.hsqldb.jdbcDriver"
    p:username="sa"
    p:password="" />
```

# Expression Language (SpEL)

- Usado en configuración **XML** o **annotation-based**
- En atributos miembro o parámetros de métodos
- **Syntax** para definir la expresión:

`{ <expression string> }`

# Fichero de propiedades (Configuración de properties)

- XML o annotation-based

```
<bean  
  class="org.springframework.beans.factory.config.PropertyPlaceholderConfigurer">  
  <property name="locations" value="classpath:usuarios.properties" />  
</bean>
```

```
<context:property-placeholder location="classpath:pruebas.properties" />
```