

Fundamentos Spring

A woman with long red hair is sitting cross-legged on a wooden floor, wearing a VR headset and a green t-shirt. She is reaching out with her right hand towards a large, curved wall display that shows an underwater scene with a shark and various fish. The scene is dimly lit, with the primary light source coming from the display. On the left side of the image, there is a large, stylized green plant with long, pointed leaves. The overall atmosphere is futuristic and immersive.

Fundamentos Spring

- 1. Spring Framework
- 2. Spring Core
- 3. Spring Data



1

Spring Framework

1. Spring Framework

Introducción



Ligero

Extendido

Mantenible

Ampliable

1. Spring Framework

Introducción

- **Spring Framework** está actualmente dividido en módulos, cada uno orientado a una finalidad concreta.
- Cada proyecto podrá utilizar los módulos que necesiten.
- Existe un Core necesario para empezar a utilizarlo.

Documentación oficial:

<http://spring.io>

1. Spring Framework

Introducción

- Spring permite desarrollar aplicaciones *flexibles*, altamente *cohesivas* y con un bajo *acoplamiento*.
- Promueve el uso de clases Java Simples (POJO – Plain Old Java Object) para la programación orientada a *interfaces* y *la configuración de servicios* (Manejo de *Transacciones*, Manejo de *Excepciones*, *Parametrización* de la aplicación).

Características Principales:

- **DI** (Dependency Injection): Este patrón de diseño permite suministrar objetos a una clase (POJO) que tiene dependencias, en lugar de ser ella misma quien los proporcione.
- **AOP** (Aspect Oriented Programming): AOP es un paradigma de programación que permite **modularizar** las aplicaciones y mejorar la separación de responsabilidades entre módulos y/o clases.

1. Spring Framework

Módulos



1. Spring Framework

Módulos



Spring Framework Runtime

Ámbito de esta formación:

Spring Data

Data Access/Integration

JDBC

ORM

OXM

JMS

Transactions

Web

WebSocket

Servlet

Web

Portlet

AOP

Aspects

Instrumentation

Messaging

Core Container

Beans

Core

Context

SpEL

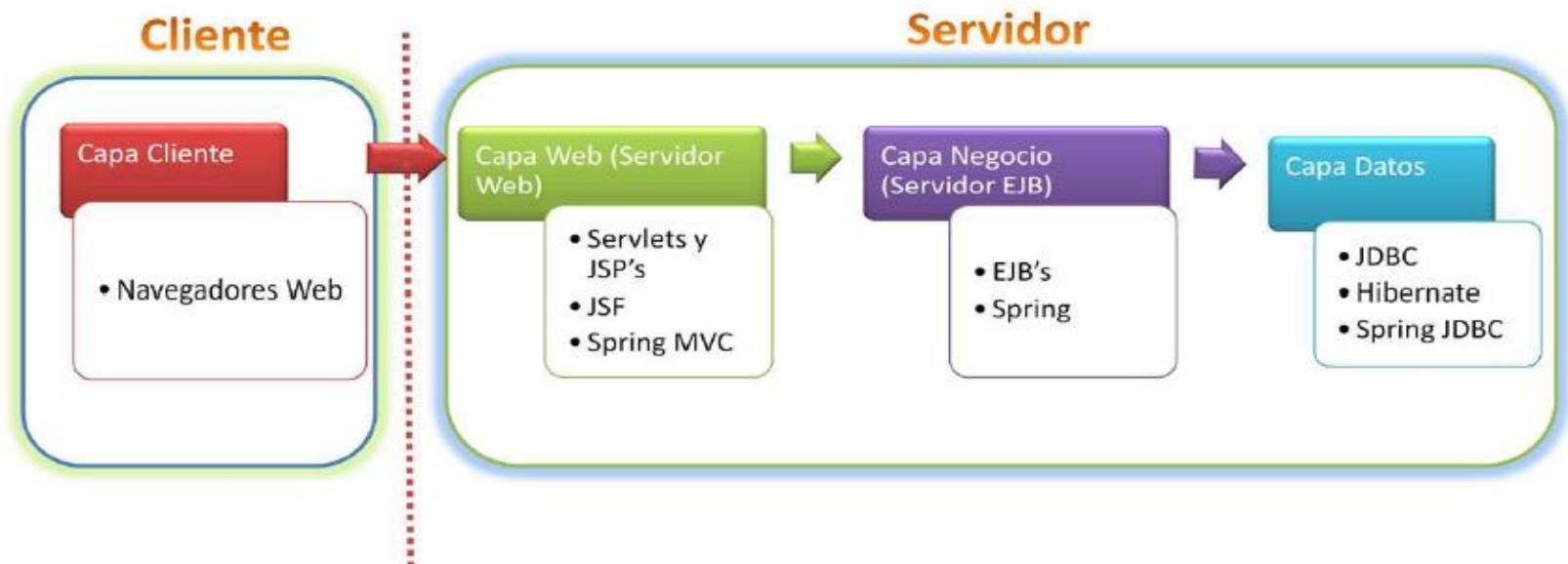
Test

- Inversión de Control (IoC)
- Inyección de Dependencia (ID)
- Spring Container (IoC Container)

1. Spring Framework

Arquitectura

Una aplicación empresarial en Java se compone de distintas **capas**, cada una con una función muy específica.








Ventajas de la separación por capas:

- Separación de responsabilidades.
- Mejor mantenimiento de la aplicación.
- Especialización de los programadores en cada capa.

1. Spring Framework

Portfolio

Spring provee un portafolio de soluciones bastante amplia:

- **Spring Web Flow** construido sobre Spring MVC para definir y gestionar flujos entre páginas. 
- **Spring-WS** permite facilitar la creación de Servicios Web basados en el intercambio de documentos (*document driven o contract first*).
- **Spring Security** módulo de seguridad para aplicaciones Web. 
- **Spring Batch** módulo para crear procesos batch, formado por una secuencia de pasos. 
- **Spring Social** provee conectividad y autorización a redes sociales como Facebook, Twitter, Google+, LinkedIn, etc. 
- **Spring Mobile** es una extensión de Spring MVC, con el objetivo de simplificar el desarrollo de aplicaciones Web móviles. 
- **Spring Roo** permite el desarrollo rápido de aplicaciones Java. 

1. Spring Framework

Recursos online

Spring tiene una comunidad de desarrolladores cada vez más extensa. Algunos de los canales que podemos utilizar para discutir sobre cualquier elemento del framework, documentarnos o pedir ayuda son:

Documentación: <https://spring.io/docs>

Proyectos: <https://spring.io/projects>

Preguntas en StackOverflow: <https://spring.io/questions>

Comunidad de Google: <https://plus.google.com/communities/101558368749171857306>

Blog: <https://spring.io/blog>



2

Spring Core

Inyección de dependencias

Inversión de Control

Contenedor

Beans

BeanFactory

ApplicationContext

Configurando Beans con XML

Configurando Beans con Anotaciones

Cargando contexto de Spring

2. Spring Core

Inyección de dependencias

La inyección de dependencias (Dependency Injection) es un patrón de diseño orientado a objetos, en el que se suministran objetos a una clase en lugar de ser la propia clase la que cree el objeto.

Libera a la clase del coste y el acoplamiento que implica la creación de los objetos dependiente.

2. Spring Core

Inversión de Control

La Inversión de control (Inversion of Control) es un método de programación en el que el flujo de ejecución de un programa se invierte respecto a los métodos de programación tradicionales.

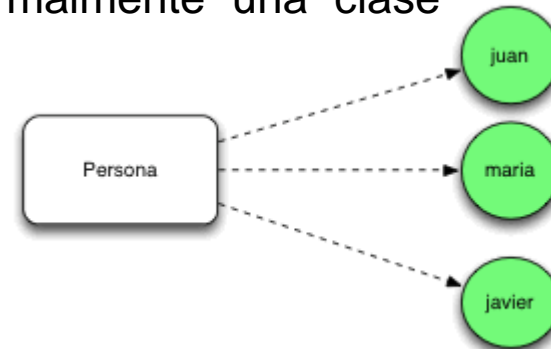
Tradicionalmente el programador especifica la secuencia de decisiones y procedimientos que pueden darse durante el ciclo de vida de un programa mediante llamadas a funciones.

En su lugar, en la inversión de control se especifican respuestas deseadas a sucesos o solicitudes de datos concretas, dejando que algún tipo de entidad o arquitectura externa lleve a cabo las acciones de control que se requieran en el orden necesario y para el conjunto de sucesos que tengan que ocurrir.

2. Spring Core

Contenedor – El patrón Singleton

Este patrón de diseño se encarga de que una clase determinada únicamente pueda tener un único objeto. Normalmente una clase puede instanciar todos los objetos que necesite.



Sin embargo una clase que siga el patrón Singleton tiene la peculiaridad de que solo puede instanciar un único objeto. Este tipo de clases son habituales en temas como configurar parámetros generales de la aplicación ya que una vez instanciado el objeto los valores se mantienen y son compartidos por toda la aplicación. La clase que va a contener nuestros objetos Singleton será el '**Contenedor**' Spring.

2. Spring Core

Contenedor

El contenedor de Spring es el espacio de memoria donde son cargados un conjunto de objetos instanciados y configurados a lo largo de toda la vida de la aplicación.

Dentro del contenedor de Spring se cargan objetos como controladores, manejadores, filtros, previene la concurrencia y optimizar los recursos, tanto a nivel de programación como de sistema, para cada llamada de uno o varios clientes vamos a reutilizar un mismo recurso de tipo concreto ya existente, en lugar de crear uno por cada llamada recibida, cada objeto en el contenedor de Spring se comporta como un Singleton.

El contenedor, además de albergar las instancias de todos estos objetos de Spring, hace posible una de las principales características de Spring, la inyección de dependencias e inversión de control.

2. Spring Core

Ejemplo de inicialización de contenedor Spring

```
public static void main(String[] args) {  
  
    ClassPathXmlApplicationContext context = new ClassPathXmlApplicationContext("application-context.xml");  
  
    try {  
        ProcesoPrincipal procesoPrincipal = context.getBean(ProcesoPrincipal.class);  
        procesoPrincipal.ejecutar();  
    } finally {  
        context.close();  
    }  
  
}
```

2. Spring Core

Bean

- Un **Bean** en Spring no es mas que un objeto configurado e instanciado en el **contenedor de Spring** usado entre otras cosas para la **inyección de dependencias**.
- Todos los *beans* permanecen en el contenedor durante toda la vida de la aplicación o hasta que nosotros los destruyamos.
- Tener los beans en el contenedor nos permite **inyectarlos** en otros beans, **reutilizarlos**, o poder **acceder a ellos** desde cualquier lugar de la aplicación en el momento que queramos.



2. Spring Core

FactoryBean

- **FactoryBean** es un patrón usado para encapsular la ***lógica de construcción*** de objetos en una clase.
- Utilizable para codificar la ***construcción de objetos complejos*** de manera reutilizable.
- Cada bean tiene un **identificador** para poder obtenerlo desde la BeanFactory.

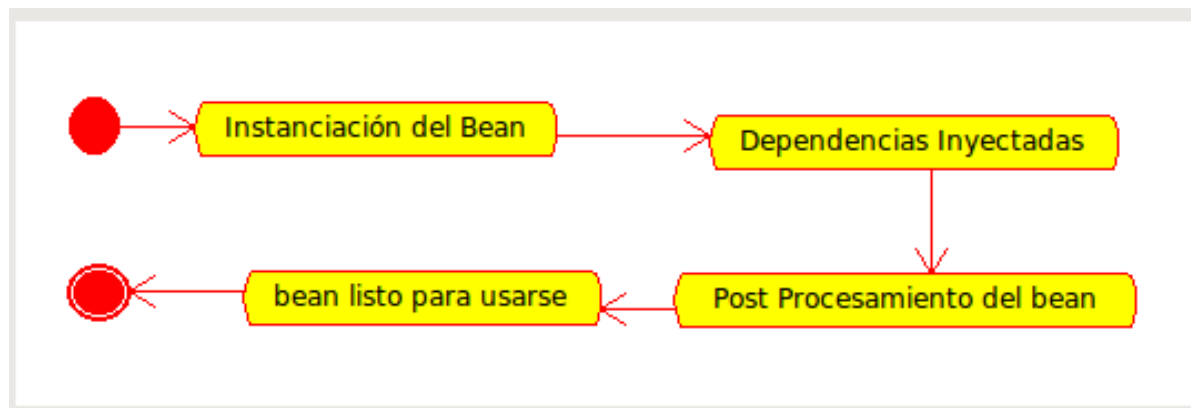
```
<!-- use the DataSource exposed by JNDI -->  
<bean id="dataSource" class="org.springframework.jndi.JndiObjectFactoryBean">  
    <property name="jndiName" value="java:comp/env/jdbc/mydb"/>  
</bean>  
<bean id="myDAO" class="com.dao.MyDAO">  
    <property name="dataSource" ref="dataSource"/>  
</bean>
```

2. Spring Core

ApplicationContext

La fase de inicialización esta completa cuando se crea el **Contexto**:

1. Parsean archivos XML de configuración.
2. Las definiciones de los beans son cargados en el contexto del BeanFactory.
3. Se invocan a clases y métodos especiales que nos permiten manipular y transformar grupos de definiciones de beans antes que los objetos sean creados (***BeanFactoryPostProcessor*** como *PropertyPlaceholderConfigurer*, *CustomScopeConfigurer*, *AspectJWeavingEnabler*...).



2. Spring Core

Configurando Beans con XML

La configuración mas básica contiene **identificador** y **clase** del objeto.

```
<bean id="idDelBean" class="LaClase" />
<bean id="idOtroBean" class="LaOtraClase" />
```

Se pueden inyectar valores de distintos tipos:

```
<bean id="idDelBean" class="LaClase">
  <property name="nombre" value="valor de cadena" />
  <property name="unEntero" value="5" />
  <property name="lista" value="5">
    <list>
      <value>valor 1</value>
      <value>valor 2</value>
    </list>
  </property>
</bean>
```

Se pueden inyectar beans dentro de otros beans:

```
<bean id="bean1" class="LaClase">
  <property name="miDependencia" ref="otroBean" />
</bean>
<bean id="otroBean" class="OtraClase" />
```

2. Spring Core

Configurando Beans con XML

Inyección en el constructor:

```
<bean id="bean1" class="LaClase">
    <constructor-arg type="java.lang.String" value="valor" />
    <constructor-arg type="java.lang.Integer" value="3" />
</bean>
```

Manejando ciclo de vida del objeto:

```
<bean id="bean1" class="LaClase" init-method="miMetodoInit" destroy-method="llamarAlFinal" />
```

Inicialización Lazy:

```
<bean id="bean1" class="LaClase" lazy-init="true" />
```

Spring Tools 3 Add-On (aka Spring Tool Suite 3) 3.9.5.RELEASE



Spring Tools 3 The Spring Tools 3 contain the previous generation Spring tooling for Eclipse, mostly focused on working with Spring apps configured using XML and... [more info](#)

by [Pivotal](#), EPL

[J2EE spring Spring IDE Cloud jee ...](#)

NTT DATA

2. Spring Core

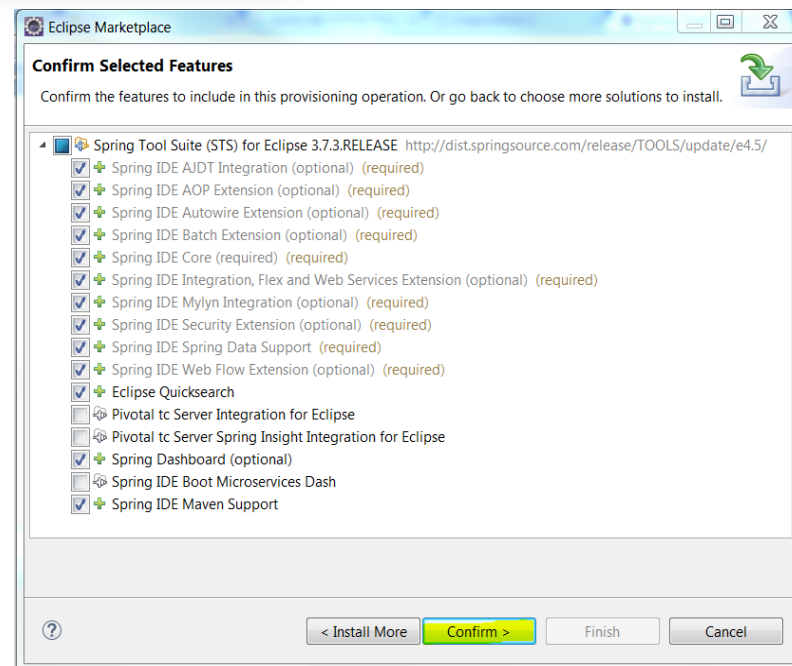
Actividades

Instalar Spring STS en eclipse y .

- Ir a <https://marketplace.eclipse.org/content/spring-tool-suite-sts-eclipse>
- Clic en botón **Install** y moverlo hacia eclipse.
- Seleccionar las opciones que nos interesan.
- Aceptar todos los términos del Contrato.
- Clic en botón Confirmar y al terminar reiniciar Eclipse.

1. Ejercicio Hola Mundo

- Crear proyecto Maven.
- Incluir librerías *spring-core*, *spring-context-support* y *spring-beans*.
- Crear clase **BeanSpring** con el atributo **mensaje** y su get y set.
- Definir fichero **applicationContext.xml** y el bean BeanSpring con un valor de mensaje por defecto.
- Crear clase main donde se imprime en consola el mensaje desde el bean.



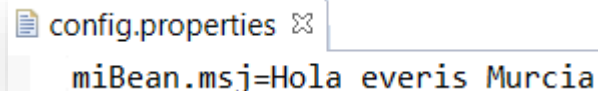
```
public static void main(String[] args) {  
    ApplicationContext context =  
        new ClassPathXmlApplicationContext("applicationContext.xml");  
    BeanFactory factory = context;  
    BeanSpring miBean = (BeanSpring) factory.getBean("miBean");  
    System.out.println("Mensaje: " + miBean.getMensaje());  
}
```


2. Spring Core

Actividades

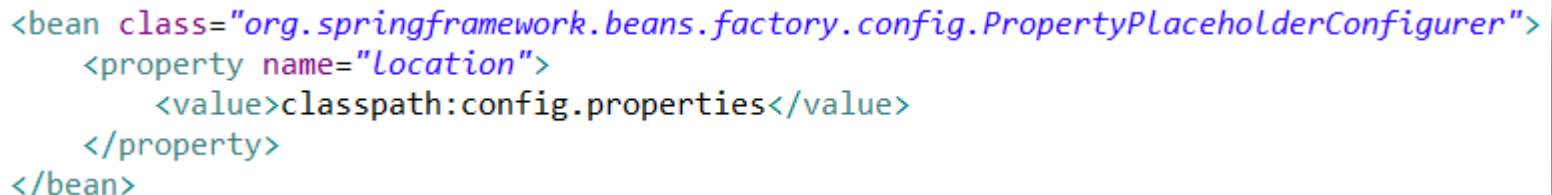
2. Uso de fichero properties para inicializar Beans

- Partiendo de la actividad anterior, vamos a modificarla para que el mensaje del 'BeanSpring' lo coja desde un fichero de properties.
- Para ello crearemos primero el fichero 'config.properties' y lo dejaremos en 'java/main/resources/'
 - El fichero contendrá únicamente:



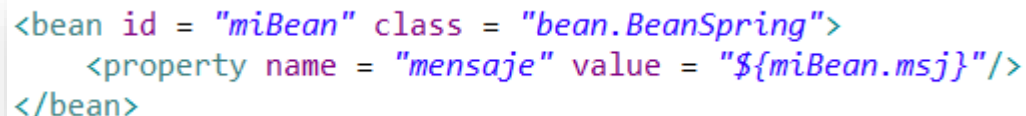
```
config.properties
miBean.msj=Hola everis Murcia
```

- En el 'applicationContext' añadiremos el siguiente código:



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

- Y por último actualizaremos el valor de la propiedad en 'miBean' y ejecutaremos la aplicación:



```
<bean id = "miBean" class = "bean.BeanSpring">
  <property name = "mensaje" value = "${miBean.msj}"/>
</bean>
```

2. Spring Core

Actividades

3. Actividad de Inyección de Dependencias con Bean Spring

- Realiza de nuevo la 'Actividad 1' utilizando Beans de Spring.
- No se debe inicializar ningún objeto en el main, ya que se usarán Bean.
- Crear proyecto Maven.
- Incluir librerías *spring-core*, *spring-context-support* y *spring-beans*.
- Definir en el fichero **applicationContext.xml** los Beans necesarios.

```
NO DI: Soy un 'Camion' con motor 'gasoil' y          capacidad de maletero '500' litros  
DI: Soy un 'Coche' con motor 'gasolina' y capacidad de maletero '200' litros
```

```
public static void main(String[] args) {  
  
    ApplicationContext context = new ClassPathXmlApplicationContext("applicationContext.xml");  
    try {  
        BeanFactory factory = context;  
  
        VehiculoDI VehiculoDI = (VehiculoDI) factory.getBean("vehiculoDI");  
        System.out.println("    DI: " + VehiculoDI.identificate());  
    } finally{  
        ((ClassPathXmlApplicationContext) context).close();  
    }  
  
}
```

2. Spring Core

Actividades

4. Proyecto Interprete con XML

- Crear proyecto Maven.
- Definir un Interfaz **Interprete** con los métodos **saludar** y **despedirse** que imprimirán en consola un mensaje específico.
- Definir la clase **TraductorEspaniol** que implementará la interfaz anterior y contenga la propiedad *nombre*, e implemente la interfaz *Interprete*.
- Definir la clase **TraductorIngles** similar a la anterior pero que mostrará los mensajes en Inglés.
- Definir fichero **applicationContext.xml** que contenga ambos beans con valores por defecto.
- Crear clase main **Actividad4** que desde el traductor ejecute los interpretes en ambos lenguajes a partir del siguiente código:

```
public static void main(String[] args) {
    ClassPathXmlApplicationContext context
        = new ClassPathXmlApplicationContext("applicationContext.xml");
    try{
        Interprete traductorEspaniol= (Interprete) context.getBean("traductorEspaniol");
        traductorEspaniol.saludar();
        traductorEspaniol.despedirse();

        Interprete traductorIngles= (Interprete) context.getBean("traductorIngles");
        traductorIngles.saludar();
        traductorIngles.despedirse();

    } finally{
        ((ClassPathXmlApplicationContext) context).close();
    }
}
```

Hola mi nombre es Yahima Duarte
Hasta pronto...!!!

Hello my name is Lucas Silva
Goodbye...!!!

```
public static void main(String[] args) {  
    ClassPathXmlApplicationContext context  
        = new ClassPathXmlApplicationContext("applicationContext.xml");  
    try{  
        Interpreter traductorEspaniol= (Interpreter) context.getBean("traductorEspaniol");  
        traductorEspaniol.saludar();  
        traductorEspaniol.despedirse();  
  
        Interpreter traductorIngles= (Interpreter) context.getBean("traductorIngles");  
        traductorIngles.saludar();  
        traductorIngles.despedirse();  
    } finally{  
        ((ClassPathXmlApplicationContext) context).close();  
    }  
}
```



```
<bean id="traductorEspaniol" class=" " >  
    <property name="nombre" value="Yahima Duarte"/>  
</bean>  
  
<bean id="traductorIngles" class=" " >  
    <property name="nombre" value="Lucas Silva"/>  
</bean>
```



Hola mi nombre es Yahima Duarte
Hasta pronto...!!!

Hello my name is Lucas Silva
Goodbye...!!!

2. Spring Core

Configurando Beans con Anotaciones

Otra manera de definir los Beans (a partir de la versión 2) es con las **Anotaciones**. Aunque no definamos los Beans con XML, este fichero siempre tiene que estar presente.

Beans con anotaciones:

- **@Service**: Para definir las componentes de negocio.
- **@Repository**: Para definir los DAO.
- **@Component**: Para componentes mas especificas.

```
package es.everis.spring.negocio;  
  
@Service("miGestor")  
public class GestorUsuarios {  
    public UsuarioTO login(String login, String password) {...}  
}
```

```
<beans>  
    <context:component-scan base-package="es.everis.spring"/>  
</beans>
```

2. Spring Core

Configurando Beans con Anotaciones

Para acceder a un Bean desde otro Bean se usa la anotación **@Autowired**.



“Programar contra interfaces, no implementaciones”

```
package es.everis.spring.service;
```

```
@Service
```

```
public class UsuariosServiceImpl implements IUsuariosService{
```

```
    @Autowired
```

```
    private IUsuariosDAO userDAO;
```

```
}
```

```
package es.everis.spring.dao;
```

```
@Repository
```

```
public class UsuariosDAOImpl implements IUsuariosDAO{
```

```
    public Usuario getUsuario(String id) {
```

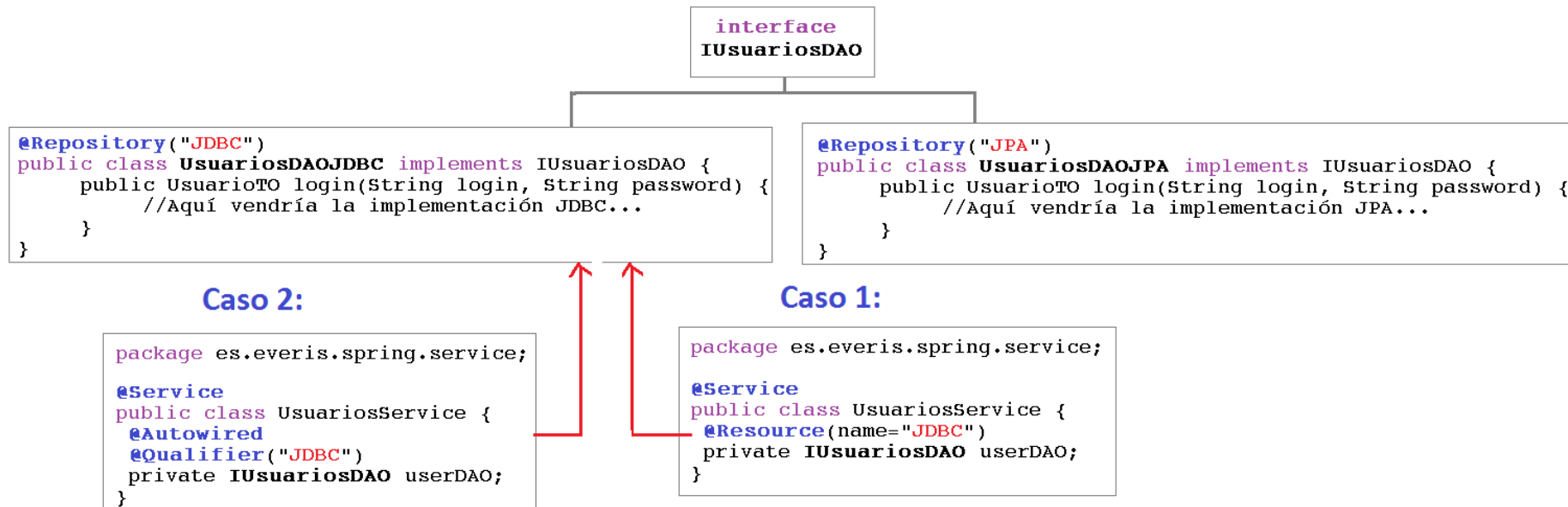
```
}
```

2. Spring Core

Configurando Beans con Anotaciones

@Autowired se puede usar solo si existe una *única implementación* de un Bean. En caso de múltiples implementaciones de un Bean se puede inyectar de dos maneras:

1. Identificándolo por nombre en la anotación.
2. Usando @Autowired + @Qualifier



2. Spring Core

Configurando Beans con Anotaciones

@Value para recoger el valor de una propiedad desde un fichero de properties:

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



config ✕
name=Rocío Moreno



```
@Value(value = "${name}")  
String nombre;
```

2. Spring Core

Configurando Beans con Anotaciones

Una vez definidas nuestras clases nuestro applicationContext.xml quedaría así:

```
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xmlns:context="http://www.springframework.org/schema/context"
       xsi:schemaLocation="http://www.springframework.org/schema/beans http://www.springframework.org/schema/beans/spring-beans.xsd
                           http://www.springframework.org/schema/context http://www.springframework.org/schema/context/spring-context-4.3.xsd">

    <context:annotation-config /> OR|
    <context:component-scan base-package="beans" />
</beans>
```

<context:annotation-config> se usa para activar las anotaciones en los beans registrados dentro del contexto de Spring.

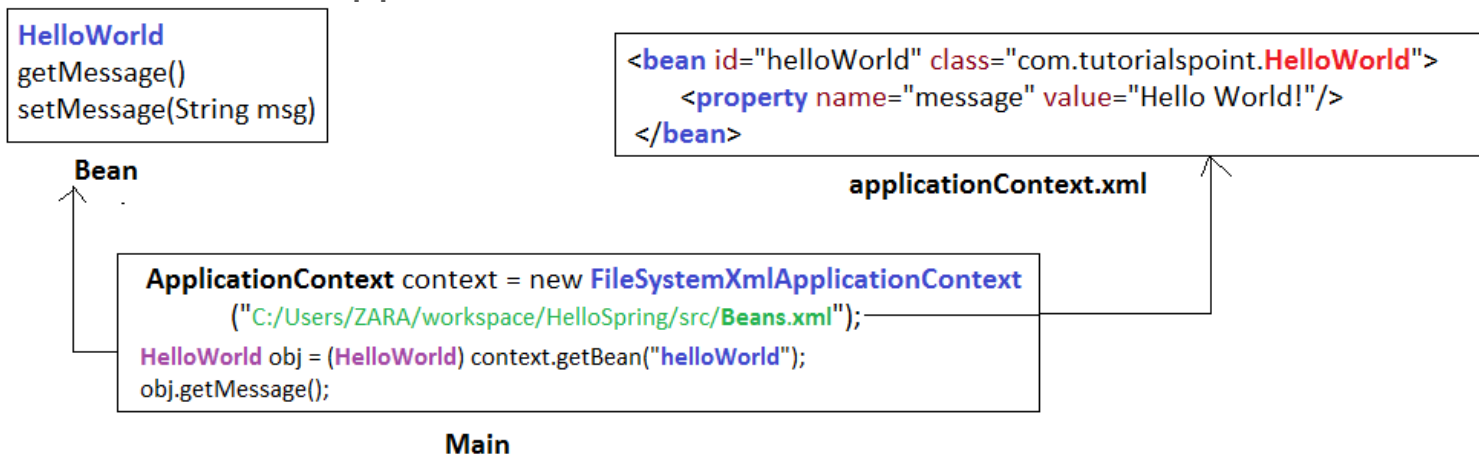
<context:component-scan> hace lo mismo de **annotation-config** pero escanea y registra los beans creados por nosotros en el paquete especificado de la aplicación.

2. Spring Core

Cargando contexto de Spring

Las 3 maneras mas usadas para cargar el contexto de Spring son:

- **FileSystemXmlApplicationContext**: Este contenedor carga las definiciones de los beans desde un archivo XML proporcionando la ruta completa.
- **ClassPathXmlApplicationContext**: Este contenedor también carga las definiciones de los beans desde un archivo XML que debe estar presente en el CLASSPATH.
- **WebXmlApplicationContext**: Este contenedor carga las definiciones de los beans desde el web application.



2. Spring Core

Cargando contexto de Spring

La manera de acceder a los Beans cambia si es una **Aplicación Web** o de escritorio.

Si nuestra aplicación web no usa el modulo de **Spring MVC**, los Servlets y JSP no son gestionados por Spring por lo que el acceso a los beans debe hacerse de esta manera:

```
//En un servlet o JSP, sin Spring MVC  
ServletContext sc = getServletContext();  
WebApplicationContext wac =  
WebApplicationContextUtils.getWebApplicationContext(sc);  
IUsuariosService service = wac.getBean(IUsuariosService.class);
```

2. Spring Core

Actividades

5. Proyecto Interprete con Anotaciones

- Realizar la actividad 7, pero definiendo los beans con anotaciones en vez de en el applicationContext.xml

```
public static void main(String[] args) {
    ClassPathXmlApplicationContext context
        = new ClassPathXmlApplicationContext("applicationContext.xml");
    try{
        Interprete traductorEspaniol= (Interprete) context.getBean("traductorEspaniol");
        traductorEspaniol.saludar();
        traductorEspaniol.despedirse();

        Interprete traductorIngles= (Interprete) context.getBean("traductorIngles");
        traductorIngles.saludar();
        traductorIngles.despedirse();

    } finally{
        ((ClassPathXmlApplicationContext) context).close();
    }
}
```



3

Spring Data

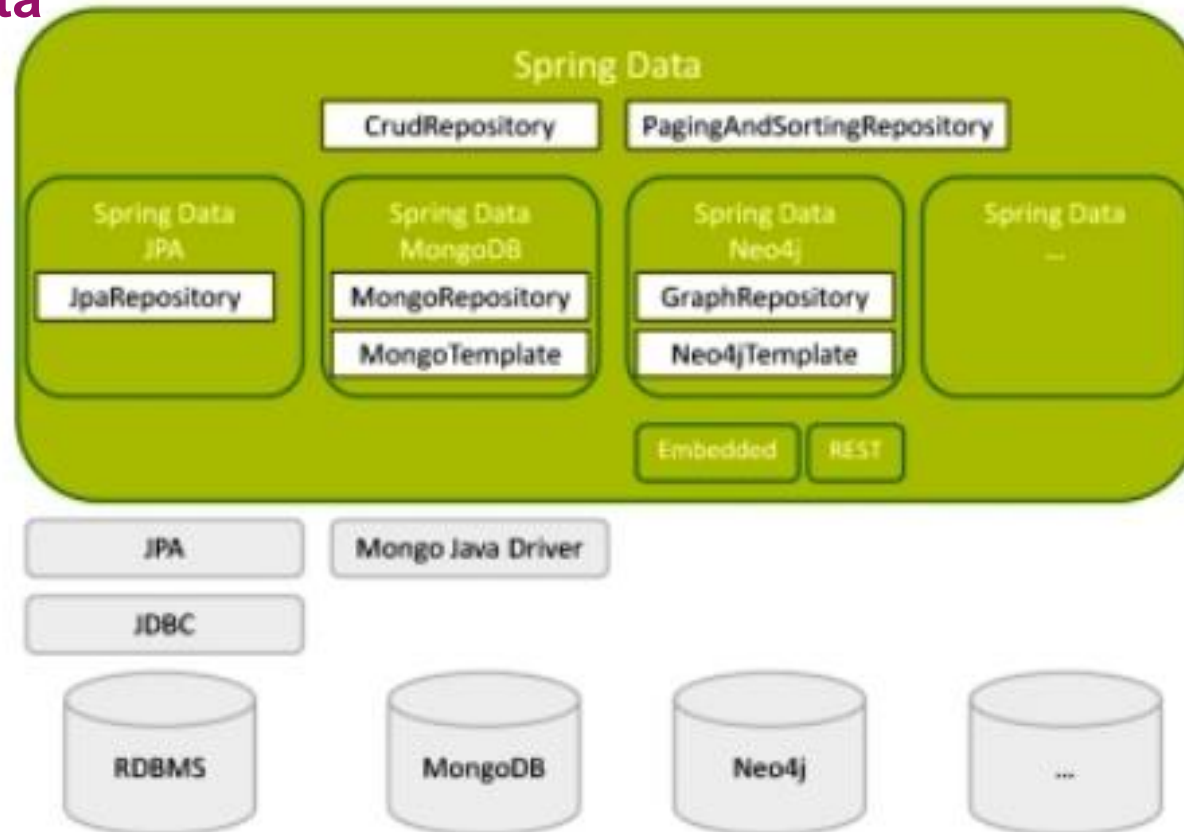
Spring Data

Spring DAO

Caché

Spring Transaction Management

Spring Data

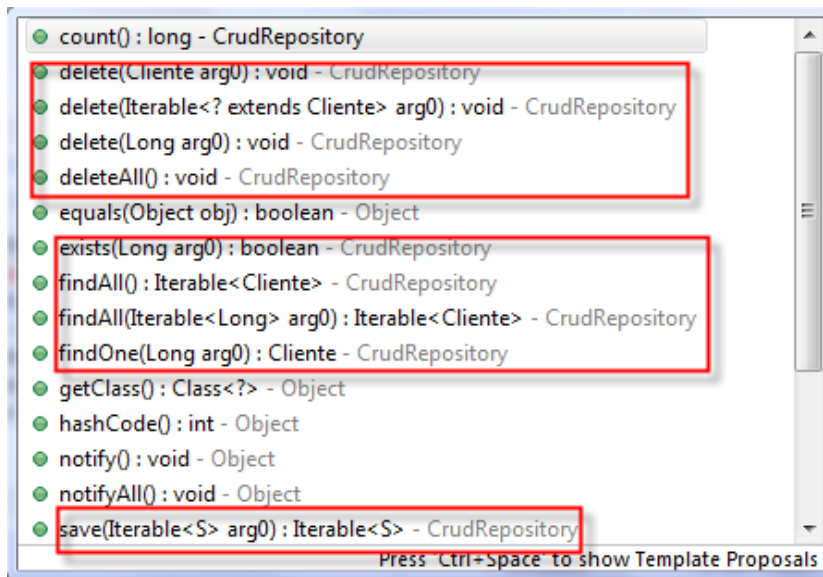


Spring Data es un proyecto de SpringSource para proporcionar un acceso fácil y unificado a diferentes clases de almacenamiento, tanto para bases de datos relacionales como almacenamientos NoSQL.

Spring Data

¿Qué necesitamos definir para trabajar con Spring Data?

1. Definir el modelo de negocio (Por ejemplo: Una clase Cliente)
2. Crear un repositorio (clase Repository, CrudRepository o PagingAndSortingRepository)



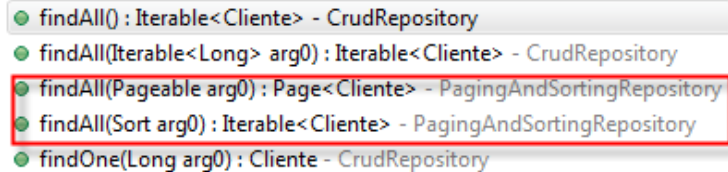
Extendiendo de `CrudRepository` obtenemos métodos automáticos de

- Inserción (save)
- Borrado (delete)
- Búsqueda (findOne y findAll)

Spring Data

¿Qué necesitamos definir para trabajar con Spring Data?

Si además queremos paginación automática de resultados podemos obtenerla extendiendo de PagingAndSortingRepository

A screenshot of a code editor showing a list of Spring Data repository methods. The methods are: findAll() : Iterable<Cliente> - CrudRepository, findAll(Iterable<Long> arg0) : Iterable<Cliente> - CrudRepository, findAll(Pageable arg0) : Page<Cliente> - PagingAndSortingRepository, findAll(Sort arg0) : Iterable<Cliente> - PagingAndSortingRepository, and findOne(Long arg0) : Cliente - CrudRepository. The last two methods are highlighted with a red rectangle.

```
• findAll() : Iterable<Cliente> - CrudRepository
• findAll(Iterable<Long> arg0) : Iterable<Cliente> - CrudRepository
• findAll(Pageable arg0) : Page<Cliente> - PagingAndSortingRepository
• findAll(Sort arg0) : Iterable<Cliente> - PagingAndSortingRepository
• findOne(Long arg0) : Cliente - CrudRepository
```

Extendiendo de PagingAndSortingRepository obtenemos métodos automáticos sobre búsqueda de entidades con criterios de paginación (Pageable) y ordenación (Sort)

- findAll (Pageable arg0)
- findAll (Sort arg0)

2. Spring Data

Spring DAO

Dentro de Spring Data, Spring provee un conjunto abstracto de clases Data Access Object (**DAO**) que hacen que el trabajo de acceso a datos con tecnologías como JDBC (*Java Database Connectivity*), JPA o Hibernate sea mucho más fácil.

- **JdbcDaoSupport**: Esta clase provee una instancia de la clase *JdbcTemplate* inicializado a partir de un *DataSource* para el acceso a datos con **JDBC**.
- **HibernateDaoSupport**: Esta clase provee una instancia de la clase *HibernateTemplate* inicializado a partir del *SessionFactory* para el acceso a datos con **Hibernate**.
- **JpaDaoSupport**: Esta clase provee una instancia de la clase *JPA_Template* inicializado a partir del *EntityManagerFactory* para el acceso a datos con **JPA**.

Spring Data

Spring DAO – **JDBC** / Hibernate / JPA

```
public UsuarioVO login(String login, String password) throws DAOException {
    Connection con=null;
    try {
        con = ds.getConnection();
        PreparedStatement ps = con.prepareStatement(SQL);
        ps.setString(1, login);
        ps.setString(2, password);
        ResultSet rs = ps.executeQuery();
        if (rs.next()) {
            UsuarioVO user = new UsuarioVO();
            user.setLogin(rs.getString("login"));
            user.setPassword(rs.getString("password"));
            user.setFechaNac(rs.getDate("fechaNac"));
            return user;
        } else return null;
    } catch(SQLException sqle) { throw new DAOException(sqle); }
    finally {
        if (con!=null) {
            try {
                con.close();
            }
            catch(SQLException sqle2) {
                throw new DAOException(sqle2);
            }
        }
    }
}
```

Spring Data

Spring DAO – **JDBC** / Hibernate / JPA

```
public class BeerDAOImpl implements BeerDAO {  
  
    private JdbcTemplate jdbcTemplate;  
  
    public Beer findById(int id) {  
        return jdbcTemplate.queryForObject("select * from Beer where id=?", new BeerMapper(), id );  
    }  
}
```

```
<?xml version="1.0" encoding="UTF-8"?>  
<beans xmlns="http://www.springframework.org/schema/beans"  
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"  
    xmlns:context="http://www.springframework.org/schema/context"  
    xsi:schemaLocation="http://www.springframework.org/schema/beans http://www.springframework.org/schema/beans/spring-beans.xsd  
        http://www.springframework.org/schema/context http://www.springframework.org/schema/context/spring-context-4.1.xsd">  
  
    <bean id = "dataSource" class = "org.springframework.jdbc.datasource.DriverManagerDataSource">  
        <property name = "driverClassName" value = "com.mysql.jdbc.Driver"/>  
        <property name = "url" value = "jdbc:mysql://10.114.88.27:3306/test" />  
        <property name = "username" value = "test" />  
        <property name = "password" value = "test" />  
    </bean>  
  
    <bean id="jdbcTemplate" class="org.springframework.jdbc.core.JdbcTemplate">  
        <property name="dataSource" ref="dataSource" />  
    </bean>  
  
    <bean id="beerDAO" class="bbdd.impl.BeerDAOImpl" >  
        <property name="jdbcTemplate" ref="jdbcTemplate" />  
    </bean>  
  
</beans>
```

Spring Data

Spring DAO – **JDBC** / Hibernate / JPA

RowMapper

```
import java.sql.ResultSet;
import java.sql.SQLException;

import org.springframework.jdbc.core.RowMapper;

public class UsuarioRowMapper implements RowMapper
{
    public Object mapRow(ResultSet rs, int rowNum) throws SQLException {
        Usuario usuario = new Usuario();
        usuario.setCustId(rs.getInt("CUST_ID"));
        usuario.setName(rs.getString("NAME"));
        usuario.setAge(rs.getInt("AGE"));
        return usuario;
    }
}
```

Spring Data

Spring DAO – **JDBC** / Hibernate / JPA

➤ SELECT con JdbcTemplate

```
jdbcTemplate.query("select * from usuarios where localidad=?", miMapper, localidad);
```

```
jdbcTemplate.queryForObject("select * from usuarios where id=?", miMapper, id);
```

➤ UPDATE con JdbcTemplate

```
jdbcTemplate.update(  
    "insert into usuarios(login, password, fechaNac) values (?, ?, ?)",  
    user.getLogin(), user.getPassword(), user.getFechaNac());
```

```
jdbcTemplate.update("update usuarios set login=? where id = ?", new  
Object[]{user.getLogin(), user.getId()});
```

```
jdbcTemplate.update("delete from beer where id = ?", user.getId());
```

Spring Data

Spring DAO – JDBC / **Hibernate** / JPA

```
@Repository
public class UsuariosDAOImpl implements IUsuariosDAO{
    private HibernateTemplate hibernateTemplate;

    public List<UsuarioVO> findAll(){
        return getHibernateTemplate().find("from Usuarios");
    }
}
```

```
<beans>
    <bean id="dataSource" class="org.apache.commons.dbcp.BasicDataSource">
        <property name="driverClassName" value="${ds.driverClassName}"></property>
        <property name="url" value="${ds.url}"></property>
        <property name="username" value="${ds.user}"></property>
        <property name="password" value="${ds.password}"></property>
    </bean>
    <bean id="mySessionFactory" class="org.springframework.orm.hibernate3.LocalSessionFactoryBean">
        <property name="dataSource" ref="dataSource"></property>
        ...
    </bean>
    <bean id="myTemplate" class="org.springframework.orm.hibernate3.HibernateTemplate">
        <property name="sessionFactory" ref="mySessionFactory"></property>
    </bean>
    <bean id="userDAO" class="es.everis.spring.IUsuariosDAO">
        <property name="hibernateTemplate" ref="myTemplate"></property>
    </bean>
</beans>
```

Spring Data

Spring DAO – JDBC / **Hibernate** / JPA

➤ SELECT con HibernateTemplate

```
getHibernateTemplate().find( "from Usuarios where id=?", userId);
```

➤ UPDATE con HibernateTemplate

```
getHibernateTemplate().update(userEntity);
```

➤ INSERT con HibernateTemplate

```
getHibernateTemplate().save(userEntity);
```

➤ DELETE con HibernateTemplate

```
getHibernateTemplate().delete(userEntity);
```


Spring Data

Spring DAO – JDBC / Hibernate / **JPA**

JPA Repository permite la definición de objetos de acceso a datos simplemente especificando una interfaz que proporciona:

- Métodos CRUD genéricos.
- Métodos de consulta a partir de su nomenclatura.
- Métodos de consulta a partir de queries JPQL o mediante queries con nombre.

Todo esto sin que sea necesario desarrollar la implementación. Spring lo hace automáticamente a partir de la interfaz.

Todos los métodos son transaccionales por defecto.

Spring Data

Spring DAO – JDBC / Hibernate / **JPA**

Métodos CRUD genéricos

JpaRepository

```
public interface UserDao
extends
JpaRepository<User, Long>
{ }
```

Method Summary	
long	count() Returns the number of entities available.
void	delete(ID id) Deletes the entity with the given id.
void	delete(Iterable<? extends I> entities) Deletes the given entities.
void	delete(I entity) Deletes a given entity.
void	deleteAll() Deletes all entities managed by the repository.
void	deleteInBatch(Iterable<I> entities) Deletes the given entities in a batch which means it will create a single Query .
boolean	exists(ID id) Returns whether an entity with the given id exists.
List<I>	findAll() Returns all instances of the type.
Page<I>	findAll(Pageable pageable) Returns a Page of entities meeting the paging restriction provided in the Pageableobject.
List<I>	findAll(Sort sort) Returns all entities sorted by the given options.
I	findOne(ID id) Retrives an entity by its primary key.
void	flush() Flushes all pending changes to the database.
List<I>	save(Iterable<? extends I> entities) Saves all given entities.
I	save(I entity) Saves a given entity.
I	saveAndFlush(I entity) Saves an entity and flushes changes instantly.

Spring Data

Spring DAO – JDBC / Hibernate / **JPA**

SpringDataJpaRepository

Equivalencia. Métodos de consulta a partir de su nomenclatura

Keyword	Sample	JPQL snippet
And	findByLastnameAndFirstname	... where x.lastname = ?1 and x.firstname = ?2
Or	findByLastnameOrFirstname	... where x.lastname = ?1 or x.firstname = ?2
Between	findByStartDateBetween	... where x.startDate between 1? and ?2
LessThan	findByAgeLessThan	... where x.age < ?1
GreaterThan	findByAgeGreaterThan	... where x.age > ?1
IsNull	findByAgeIsNull	... where x.age is null
IsNotNull,NotNull	findByAge(Is)NotNull	... where x.age not null
Like	findByFirstnameLike	... where x.firstname like ?1
NotLike	findByFirstnameNotLike	... where x.firstname not like ?1
OrderBy	findByAgeOrderByLastnameDesc	... where x.age = ?1 order by x.lastname desc
Not	findByLastnameNot	... where x.lastname <> ?1
In	findByAgeIn(Collection<Age> ages)	... where x.age in ?1
NotIn	findByAgeNotIn(Collection<Age> age)	... where x.age not in ?1

```
public interface UserRepository extends JpaRepository<User, Long> {
    List<User> findByLastname(String lastname);
}
```

Spring Data

Spring DAO – JDBC / Hibernate / JPA

SpringDataJpaRepository

Métodos de consulta a partir de su nomenclatura

Es posible generar métodos de consulta para las propiedades añadiéndolos en el repositorio.

Se basa en el siguiente algoritmo

- La nomenclatura del método de consulta debe ser
 - `findByPropiedad1AndPropiedad2...AndPropiedadN`
 - Ej: `List<Cliente> findByNombre(String nombre);`
`List<Cliente> findByNombreAndApellidos(String nombre, String apellidos);`
- En caso de realizar una ejecución sobre una propiedad que o existe se obtiene una excepción en tiempo de ejecución

`org.springframework.data.mapping.PropertyReferenceException`: No property **nombres** found for type `com.everis.ejemploSpringJpaRepository.model.Cliente`

- Pueden realizarse consultas no case sensitive (añadiendo sufijo `IgnoreCase`) y por búsqueda parcial (Like).
 - Ej: `findByNombreIgnoreCase(String nombre);` o `findByNombreLike(String nombre);`

PageRequest

Constructor Summary
PageRequest (int page, int size) Creates a new PageRequest .
PageRequest (int page, int size, Sort.Direction direction, String ... properties) Creates a new PageRequest with sort parameters applied.
PageRequest (int page, int size, Sort sort) Creates a new PageRequest with sort parameters applied.

Sort

Constructor Summary
Sort (List < Sort.Order > orders) Creates a new Sort instance.
Sort (Sort.Direction direction, List < String > properties) Creates a new Sort instance.
Sort (Sort.Direction direction, String ... properties) Creates a new Sort instance.
Sort (Sort.Order ... orders)
Sort (String ... properties) Creates a new Sort instance.

Constructor Summary
Sort.Order (Sort.Direction direction, String property) Creates a new Sort.Order instance.
Sort.Order (String property) Creates a new Sort.Order instance.

Page

Method Summary	
List<T>	getContent() Returns the page content as List .
int	getNumber() Returns the number of the current page.
int	getNumberOfElements() Returns the number of elements currently on this page.
int	getSize() Returns the size of the page.
Sort	getSort() Returns the sorting parameters for the page.
long	getTotalElements() Returns the total amount of elements.
int	getTotalPages() Returns the number of total pages.
boolean	hasContent() Returns whether the Page has content at all.
boolean	hasNextPage() Returns if there is a next page.
boolean	hasPreviousPage() Returns if there is a previous page.
boolean	isFirstPage() Returns whether the current page is the first one.
boolean	isLastPage() Returns whether the current page is the last one.
Iterator<T>	iterator()

```
public static void main(String[] args) {
    try {
        context = new ClassPathXmlApplicationContext("applicationContext.xml");
        BeerDAO dao = (BeerDAO) context.getBean("beerDAO");

        // Imprimir total de cervezas:
        System.out.println( "Total cervezas: " + dao.findAll().size() );

        // Consultar cerveza con id=1 :
        Beer beer = dao.findById( 1 );
        System.out.println("Cerveza (1): "+beer);

        //Crear una nueva cerveza
        beer.setId(4);
        beer.setName("Negra");
        beer.setDescription("Mi cerveza inglesa");
        beer.setPrice(2);
        beer.setProductid("black beer");

        //Insertar cerveza nueva:
        int result = dao.insert(beer);
        System.out.println("Cervezas insertadas nuevas: "+result);

        // Consultar cerveza con id=4 :
        beer = dao.findById( 4 );
        System.out.println("Cerveza (4): "+beer);

        //Modificar cerveza nueva:
        beer.setName("Negra (black)");
        beer.setDescription("My english beer");
        result = dao.update(beer);
        System.out.println("Cervezas actualizadas: "+result);

        // Consultar cerveza con id=4 :
        beer = dao.findById( 4 );
        System.out.println("Cerveza (4): "+beer);

        //Eliminamos cerveza :
        result = dao.delete(4);
        System.out.println("Cervezas eliminadas: "+result);

        // Imprimir total de cervezas:
        System.out.println( "Total cervezas: " + dao.findAll().size() );
    } finally{
        ((ClassPathXmlApplicationContext) context).close();
    }
}
```

Spring Data

Validaciones de Beans

De forma transparente en la petición, mediante anotaciones JSR-303 en el objeto de dominio, obteniendo los resultados de conversión y validación en BindingResult.

JSR-303

```
public class JavaBean {  
    @NotNull  
    @Max(5)  
    private Integer number;  
  
    @NotNull  
    @DateTimeFormat(iso=ISO.DATE)  
    private Date date;  
}
```


Spring Data

Validaciones JSR-303

Annotation	Supported data types
@AssertFalse	Boolean, boolean
@AssertTrue	Boolean, boolean
@DecimalMax	BigDecimal, BigInteger, String, byte, short, int, long and the respective wrappers of the primitive types. Additionally supported by HV: any sub-type of Number.
@DecimalMin	BigDecimal, BigInteger, String, byte, short, int, long and the respective wrappers of the primitive types. Additionally supported by HV: any sub-type of Number.
@Digits(integer=, fraction=)	BigDecimal, BigInteger, String, byte, short, int, long and the respective wrappers of the primitive types. Additionally supported by HV: any sub-type of Number.
@Future	java.util.Date, java.util.Calendar; Additionally supported by HV, if the Joda Time date/time API is on the class path: any implementations of ReadablePartial and ReadableInstant.
@Max	BigDecimal, BigInteger, byte, short, int, long and the respective wrappers of the primitive types. Additionally supported by HV: String (the numeric value represented by a String is evaluated), any sub-type of Number.
@Min	BigDecimal, BigInteger, byte, short, int, long and the respective wrappers of the primitive types. Additionally supported by HV: String (the numeric value represented by a String is evaluated), any sub-type of Number.
@NotNull	Any type
@Null	Any type
@Past	java.util.Date, java.util.Calendar; Additionally supported by HV, if the Joda Time date/time API is on the class path: any implementations of ReadablePartial and ReadableInstant.
@Pattern(regex=, flag=)	String
@Size(min=, max=)	String, Collection, Map and arrays
@Valid	Any non-primitive type

http://docs.jboss.org/hibernate/validator/4.2/reference/en-US/html_single/#validator-defineconstraints-built-in

Spring Data

Validaciones

Otras validaciones hibernate

Annotation	Supported data types
@CreditCardNumber	String
@Email	String
@Length(min=, max=)	String
@NotBlank	String
@NotEmpty	String, Collection, Map and arrays
@Range(min=, max=)	BigDecimal, BigInteger, String, byte, short, int, long and the respective wrappers of the primitive types
@SafeHtml(whitelistType=, additionalTags=)	CharSequence
@ScriptAssert(lang=, script=, alias=)	Any type
@URL(protocol=, host=, port=, regexp=, flags=)	String

http://docs.jboss.org/hibernate/validator/4.2/reference/en-US/html_single/#validator-defineconstraints-built-in

Anotaciones de validación a la medida

Definición de la anotación con `@Constraint(validatedBy={})` y de la clase que implementa `ConstraintValidator` con método `isValid()`.

<http://docs.jboss.org/hibernate/validator/4.0.1/reference/en/html/validator-customconstraints.html>

Spring Data

Caché

La configuración de caché está basada en Ehcache mediante las siguientes anotaciones:

- `@Cacheable`: marca un método como «cacheable», asignando un nombre a la caché. Los resultados de posteriores invocaciones con los mismos parámetros serán tomados de caché.
- `@CacheEvict` / `@CachePut`: marca los métodos que podrán realizar eliminaciones/actualizaciones de la caché. Debe indicarse el nombre de la caché a la que afecta. El primero permite indicar si se eliminan todas las entradas (lo usual, por rendimiento).

@Cacheable y @CacheEvict

```
@Cacheable(value = "records")
@RequestMapping(value = "/getall", method = RequestMethod.POST)
public @ResponseBody JqgridTableDto<Event> getall() { ... }
```

```
@CacheEvict(value = "records", allEntries=true)
@RequestMapping(value = "/add", method = RequestMethod.POST)
public @ResponseBody ResponseDto<Event> add(Event event) { ... }
```

Spring Data

Spring Transaction Management

Una de las razones más importantes por la que se usa Spring es por el soporte de **transacciones**. Provee un modelo consistente de programación a través de librerías de transacciones como **JTA**, **JDBC**, **Hibernate**, **JPA** y **JDO** las cuales deben empezar y terminar a nivel de *Servicio*, **NUNCA** a nivel de DAO.

Spring provee soporte para la administración de transacciones de dos tipos:

- **Declarativa:** Spring provee distintas formas para definir este tipo de transacciones
La más habitual es utilizando anotaciones **@Transactional** (desde Spring 2.0).
- **Programáticas:** Usa interfaces abstractas de Spring para crear tu propio código y manejar las transacciones.

Spring Data

Configuración de transacciones

Ejemplo:

@Transactional

```
@Transactional(propagation=Propagation.SUPPORTS, readOnly=true)
Public class ServicioDeAdministracionImpl implements ServicioDeAdministracion {

    @Transactional(propagation=Propagation.REQUIRED, readOnly=false)
    public void guardarUsuario(Usuario usuario){ ... }

}
```

A nivel de clase se puede configurar un tipo de transacción común a todos sus métodos.

A nivel de método se pueden configurar o matizar las características de la transacción.

Spring Data

Spring Transaction Management

Transaccionalidad declarativa

- Normalmente se gestiona desde la capa de negocio, aunque está íntimamente ligada al acceso a datos.
- Lo primero que necesitamos en Spring para declararlas es un “***Transaction Manager***”. Hay varias implementaciones, dependiendo del API usado por los DAOs.

```
<jee:jndi-lookup id="miDataSource" jndi-name="DStestSpring" resource-ref="true" />

<!-- Elegimos un tipo de "Transaction Manager" (aquí para JDBC) -->
<bean id="miTxManager" class="org.springframework.jdbc.datasource.DataSourceTransactionManager">
  <property name="dataSource" ref="miDataSource"/>
</bean>

<!-- Decimos que para este Transaction Manager vamos a usar anotaciones -->
<tx:annotation-driven transaction-manager="miTxManager"/>
```

Spring Data

Spring Transaction Management

La anotación @Transactional

- Colocada delante de un método, lo hace transaccional. Delante de la clase hace que TODOS los métodos lo sean.
- El comportamiento por defecto es **rollback** automático ante excepción no comprobada (recordemos que `DataAccessException` lo es).

```
@Service
public class UsuariosServiceImpl implements IUsuariosService {
    @Autowired
    private UsuariosDAOImpl userDAO;

    @Transactional
    public void registrar(UsuarioVO user) {
        userDAO.registrarEnBD(user);
        userDAO.registrarEnListasDeCorreo(user);
    }
}
```

