Fundamentos Spring (Spring Core y Spring Data)

Orientado a profesionales que están utilizando Spring (Core/Data) sin tener los conocimientos asentados.

ever FUTURE

Fundamentos Spring (Spring Core y Spring Data)

everis (an NTT DATA Company)

Fundamentos Spring

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

ever FUTURE

1 Spring Framework

Introducción
Módulos
Arquitectura
Portfolio
Recursos online

1. Spring Framework

Introducción

1. Spring Framework

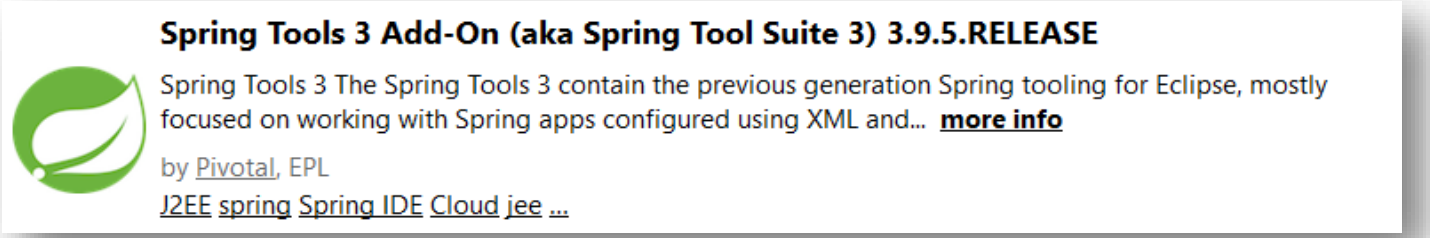
Introducción

- **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

- Nosotros vamos a necesitar:



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 Inyection): 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

Ámbito de esta formación:

Spring Framework Runtime

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

ever FUTURE

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

Inyección de dependencias. Ejemplo no DI

```
public class VehiculoNoDi {

private String tipoVehiculo;
private Motor motor;
private Maletero maletero;

public VehiculoNoDi(String tipoVehiculo, String tipoMotor, int capacidadMaletero) {
 this.tipoVehiculo = tipoVehiculo;
 this.motor = new Motor(tipoMotor);
 this.maletero = new Maletero(capacidadMaletero);
}

public String identificate() {
 return "Soy un '" + tipoVehiculo + "' con motor '" + motor.getTipo() + "' y
capacidad de maletero '" + maletero.getCapacidad() + "' litros";
}

}
```

2. Spring Core

Inyección de dependencias. Ejemplo no DI

El uso de Vehículo exige que se le pasen todos los parámetros necesarios para realizar la construcción de forma interna.

```
public static void main(String[] args) {
    VehiculoNoDi vehiculo = new VehiculoNoDi("Camion", "gasoil", 500);

    System.out.println(vehiculo.identificate());
}
```

2. Spring Core

Inyección de dependencias. Ejemplo DI

```
public class VehiculoDI {

private String tipoVehiculo;
private Motor motor;
private Maletero maletero;

public VehiculoDI(String tipoVehiculo, Motor motor, Maletero maletero) {
    this.tipoVehiculo = tipoVehiculo;
    this.motor = motor;
    this.maletero = maletero;
}

public String identificate() {
    return "Soy un '" + tipoVehiculo + "' con motor '" + motor.getTipo() + "' y
capacidad de maletero '" + maletero.getCapacidad() + "' litros";
}

}
```

2. Spring Core

Inyección de dependencias. Ejemplo DI

El uso de Vehículo no exige que se le pasen todos los parámetros. Recibe la dependencia ya formada, dejando el coste de la construcción fuera del Vehículo.

```
public static void main(String[] args) {

 Motor motor = new Motor("gasolina");
 Maletero maletero = new Maletero(200);

 VehiculoDI vehiculo = new VehiculoDI("Coche", motor,   maletero);

 System.out.println(vehiculo.identificate());
}
```

2. Spring Core

Inyección de dependencias. **Actividad 1**

Modificar el constructor de Motor añadiendo otra característica como “cilindrada”.

Comprobar como el cambio tiene impacto en VehiculoNoDi y no tiene impacto en VehiculoDI.

Menor impacto ante cambios (menor acoplamiento) es lo que se busca con el patrón de Inyección de dependencias.

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

Inversión de Control. Ejemplo no IOC

Se basa en un interprete de comando para crear Vehículos (coche, camión, moto)

```
public class NoIOC {

public void proceder() {
EntradaComando entradaComando = new EntradaComando();

boolean pedirEntrada = true;
while (pedirEntrada) {
String comando = entradaComando.recogerComando();

switch (comando) {
  case "coche":
    crearCoche();
    break;
  case "camion":
}
}
```

2. Spring Core

Inversión de Control. Ejemplo no IOC

Se basa en un interprete de comando para crear Vehículos (coche, camión, moto)

```
private void crearCoche() {
Motor motor = new Motor("gasolina");
Maletero maletero = new Maletero(200);
VehiculoDI vehiculo = new VehiculoDI("Coche", motor, maletero);
System.out.println("Creado --> " + vehiculo.identificate());
}
… …
}
```

2. Spring Core

Inversión de Control. Ejemplo IOC

Se basa en un interprete de comando para crear Vehículos (coche, camión, moto)

```
public class IOC {
    /*---*/
    public void proceder() {
        EntradaComando entradaComando = new EntradaComando();

        boolean pedirEntrada = true;
        while (pedirEntrada) {
         String comando = entradaComando.recogerComando();

         boolean encontrado = false;
         for (EscuchadorComando escuchadorComando : escuchadorSet) {
         if (escuchadorComando.getNombreComando().equals(comando)) {
                  encontrado = true;
         escuchadorComando.accionComando();
         }
         }…
```

2. Spring Core

Inversión de Control. Ejemplo IOC

Los vehículos son añadidos como escuchadores

IOC no contiene todo el código. La parte correspondiente a la ejecución de los comandos (ComandoCoche, ComandoMoto, ComandoCamion) es externa al proceso principal y será inyectada para que sea invocada desde este.

Que un proceso principal invoque código aportado desde fuera es la base de la Inversión de control

```
public class TestIOC {
        public static void main(String[] args) {
                IOC ioc = new IOC();

                ioc.annadirEscuchador(new ComandoCoche());
                ioc.annadirEscuchador(new ComandoMoto());
                ioc.annadirEscuchador(new ComandoCamion());

                ioc.proceder();
        }
}
```

2. Spring Core

Inversión de Control. **Actividad 2**

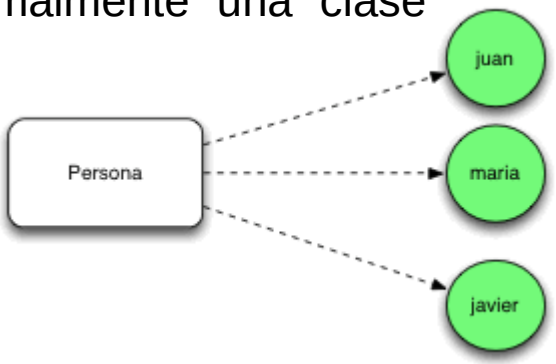
Añadir un nuevo comando para la creación de un vehículo (Tractor)

Analizar el impacto que tiene en código dicho cambio en ambos ejemplos (IOC/NoIOC).

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 =  zontext.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.

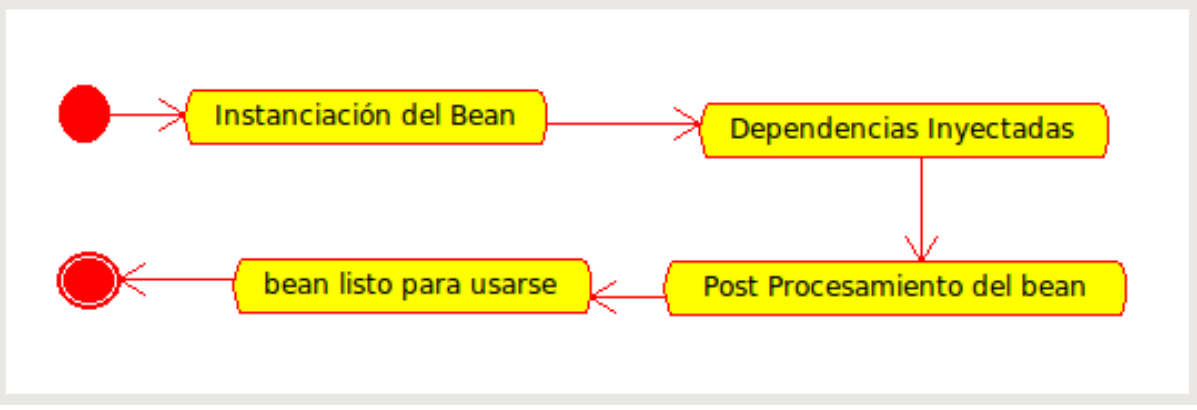
```
<!-- use the DataSource exposed by JNDI -->
<bean id="dataSource" class="org.springframework.jndi.JndiObjectFactoryBean">
    <property name="jndiName" value="java:comp/env/jdbc/mydb"/>
</bean>
<bean id="myDAO" class="com.dao.MyDAO">
    <property name="dataSource" ref="dataSource"/>
</bean>
```

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:

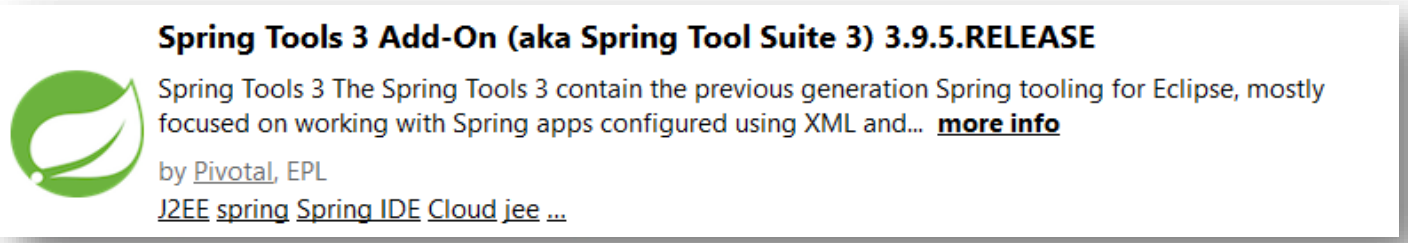
```
<bean id="bean1" class="LaClase">
    <constructor-arg type="java.lang.String" value="valor" />
    <constructor-arg type="java.lang.Integer" value="3" />
</bean>
```

Manejando ciclo de vida del objeto:

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

Inicialización Lazy:

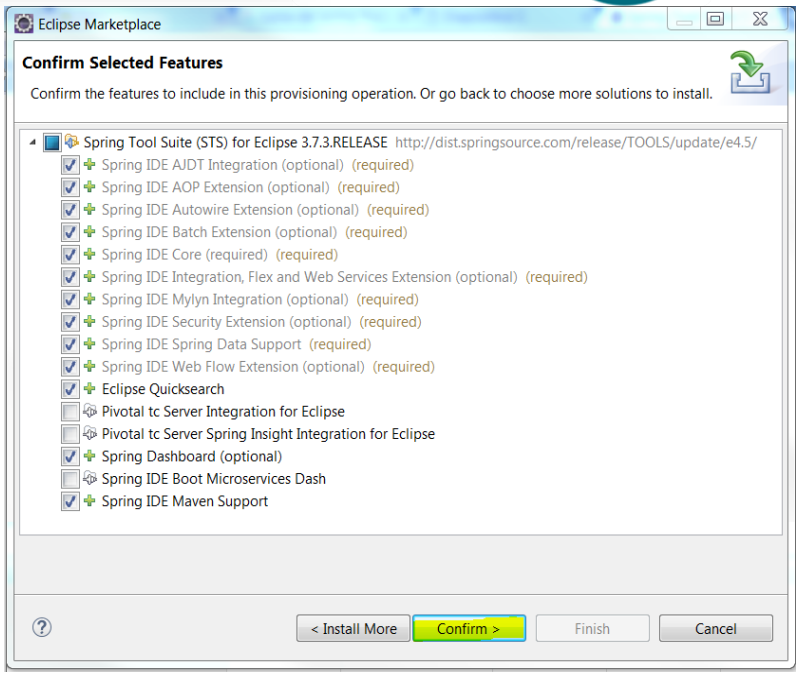
```
<bean id="bean1" class="LaClase" lazy-init="true" />
```



ever FUTURE

2. Spring Core

Actividades



3. 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.

4. Ejercicio Hola Mundo

- Crear proyecto Maven.
- Incluir librerías *spring-core, spring-context-support , spring-test y junit*.
- Crear clase **BeanSpring** con variable mensaje, 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());
}
```

5. 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>
```

6. 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 , spring-test y junit*.
- Definir en el fichero **applicationContext.xml** los Beans necesarios.

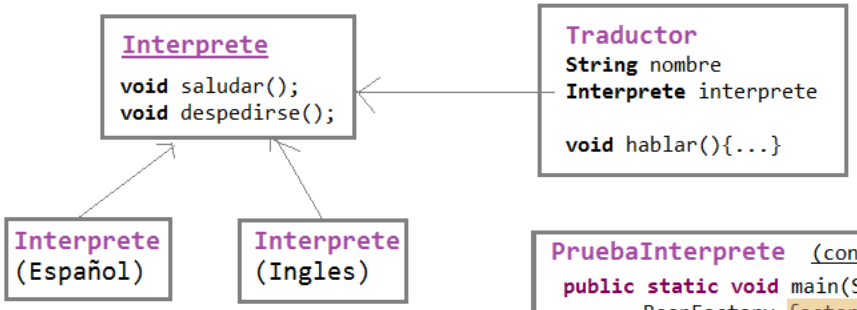
```
NO DI: Soy un 'Camion' con motor 'gasoil' y       capacidad de maletero '500' litros
   DI: Soy un 'Coche' con motor 'gasolina' y capacidad de maletero '200' litros
```

2. Spring Core

Actividades

7. Proyecto Interprete con XML

- Crear proyecto Maven **spring-interprete-xml**.
- Definir Interfaz con los métodos **saludar** y **despedirse** que imprimirán en consola un mensaje especifico.
- Definir clase **Traductor** que contenga la propiedad *nombre*, una instancia de un *interprete* y el método *hablar* que lo que hace es imprimir el saludo, luego imprime el nombre y luego imprime la despedida en el lenguaje del interprete seleccionado.
- Definir fichero **applicationContext.xml** que contenga ambos beans con valores por defecto.
- Crear clase main **PruebaInterprete** que desde el traductor ejecute los interpretes en ambos lenguajes.




```
PruebaInterprete  (con XML)
public static void main(String[] args) {
      BeanFactory factory =
         new ClassPathXmlApplicationContext("applicationContext.xml");
      Traductor traductorES = (Traductor) factory.getBean("traductorEspanol");
      traductorES.hablar();

      Traductor traductorEN = (Traductor) factory.getBean("traductorIngles");
      traductorEN.hablar();
}
```

Consola

InterpreteEspañol	InterpreteIngles
Hola mi nombre es Yahima Duarte Hasta pronto...!!!	Hello my name is Yahima Duarte 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;
}
```

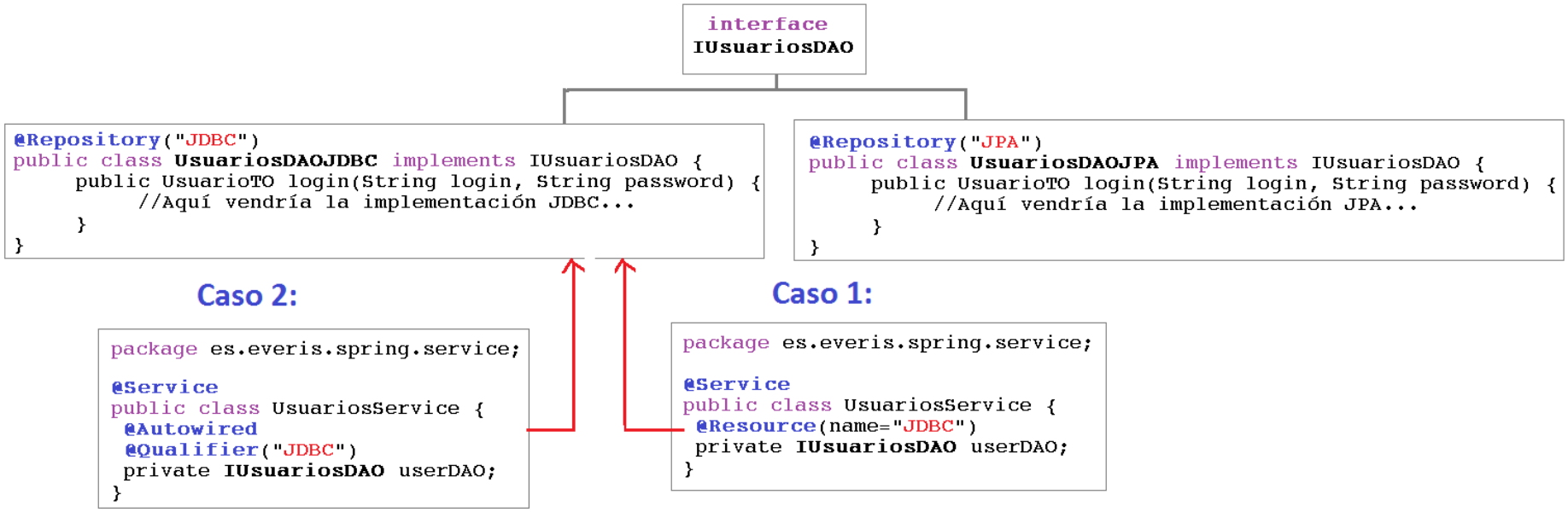
```
package es.everis.spring.dao;

@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

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.

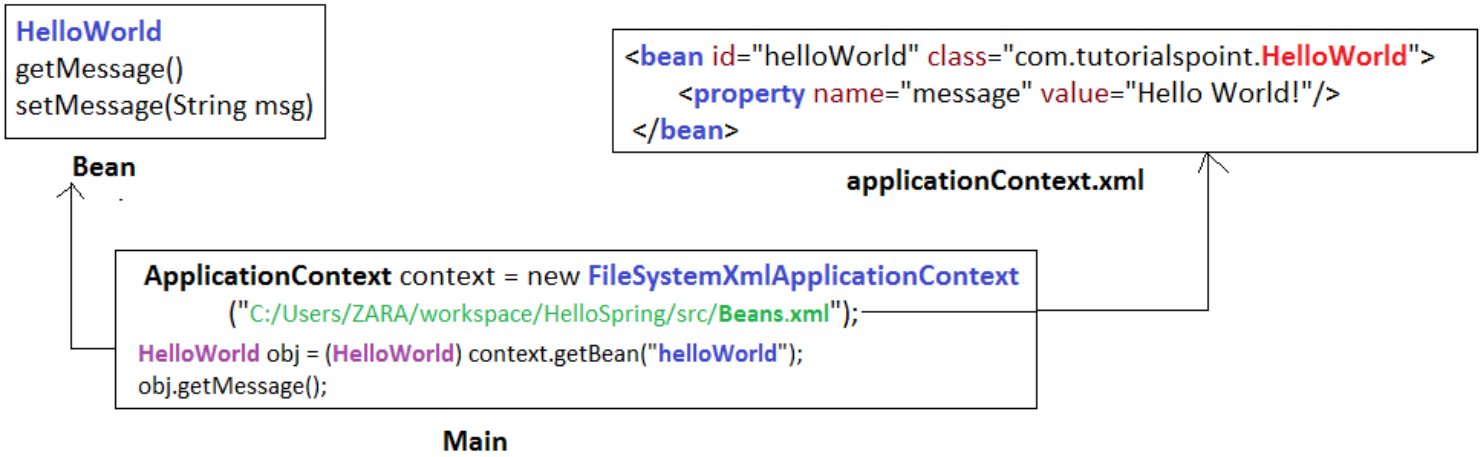
<context:component-scan> hace lo mismo de **annotation-config** pero escanea y registra los beans creados por nosotros en el paquete especificado de la aplicación.

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.



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:

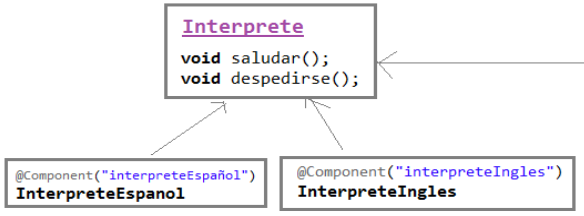
```
//En un servlet o JSP, sin Spring MVC
ServletContext sc = getServletContext();
WebApplicationContext wac =
WebApplicationContextUtils.getWebApplicationContext(sc);
IUsuariosService service = wac.getBean(IUsuariosService.class);
```

2. Spring Core

Actividades

8. Proyecto Interprete con Anotaciones

- Crear proyecto Maven **spring-interprete-annotations**.
- Definir Interfaz con los métodos **saludar** y **despedirse** que imprimirán en consola un mensaje especifico.
- Definir clase **Traductor** que contenga la propiedad *nombre*, una instancia de una clase **InterpreteFactory** que al pasar un lenguaje te devuelve un Interprete especifico y el método *hablar* que lo que hace es imprimir el saludo, luego imprime el nombre y luego imprime la despedida en el lenguaje del interprete seleccionado.
- Definir fichero **applicationContext.xml** que contenga la configuración para ***escanear*** los beans dentro del paquete "beans".
- Crear clase main **PruebaInterprete** desde donde se obtenga una instancia de los beans ***TraductorEspañol*** y ***TraductorIngles*** y al final ejecutar el método **hablar** de ambos.




```
Traductor
InterpreteFactory interpreteFactory;
String nombre;

void hablar() { ... }
Interprete getInterprete();
```

```
@Component
InterpreteFactory {
        Interprete interpreteEspañol;
        Interprete interpreteIngles;

        Interprete getInterprete(language){...}
}
```

```
PruebaInterprete  (con XML)
public static void main(String[] args) {
        BeanFactory factory = new ClassPathXmlApplicationContext("applicationContext.xml");
        TraductorEspañol traductorES = (TraductorEspañol) factory.getBean("traductorEspañol");
        traductorES.hablar();
        System.out.println();

        TraductorIngles traductorEN = factory.getBean(TraductorIngles.class);
        traductorEN.hablar();
}
```

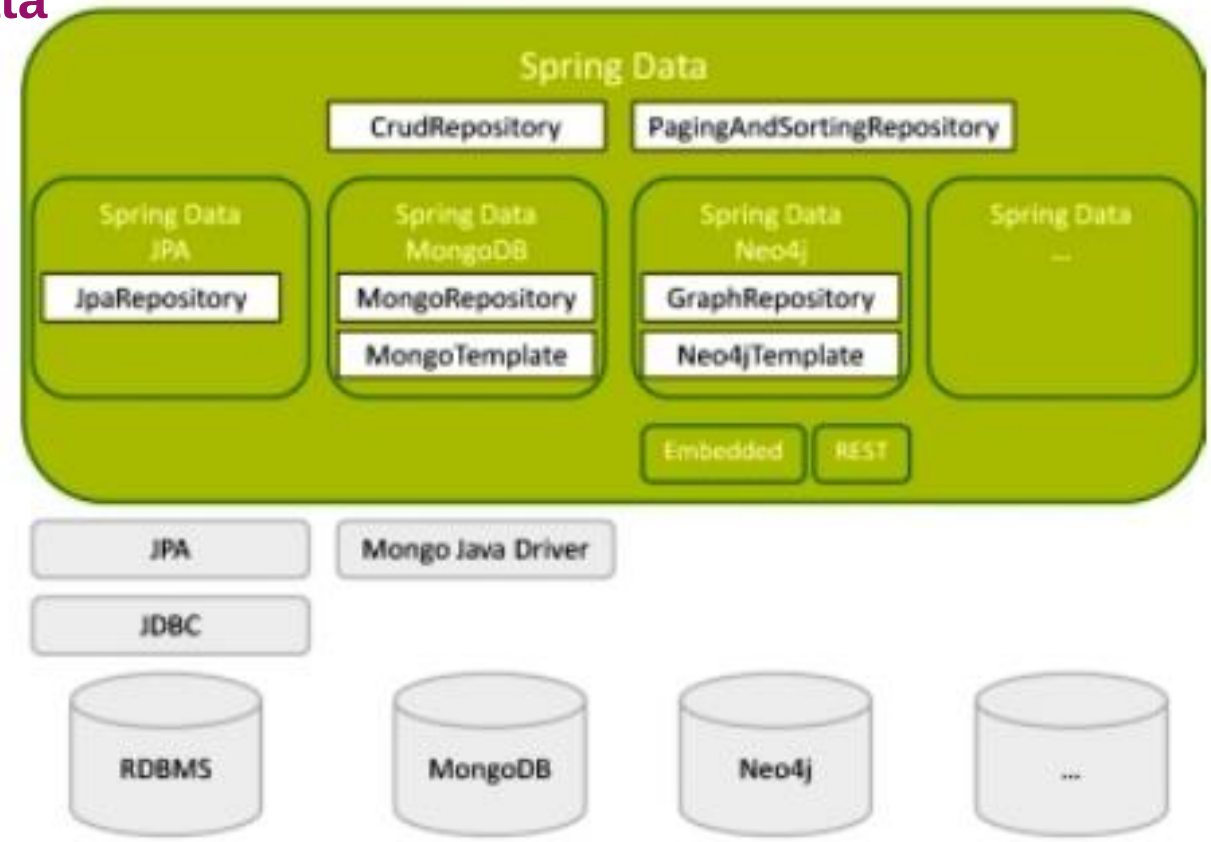
Consola

InterpreteEspañol	InterpreteIngles
Hola mi nombre es Yahima Duarte	Hello my name is Yahima Duarte
Hasta pronto...!!!	Goodbye ...!!!

3 Spring Data

Spring Data
Spring DAO
Spring Transaction Management
Ejercicios
Spring Data

Spring Data

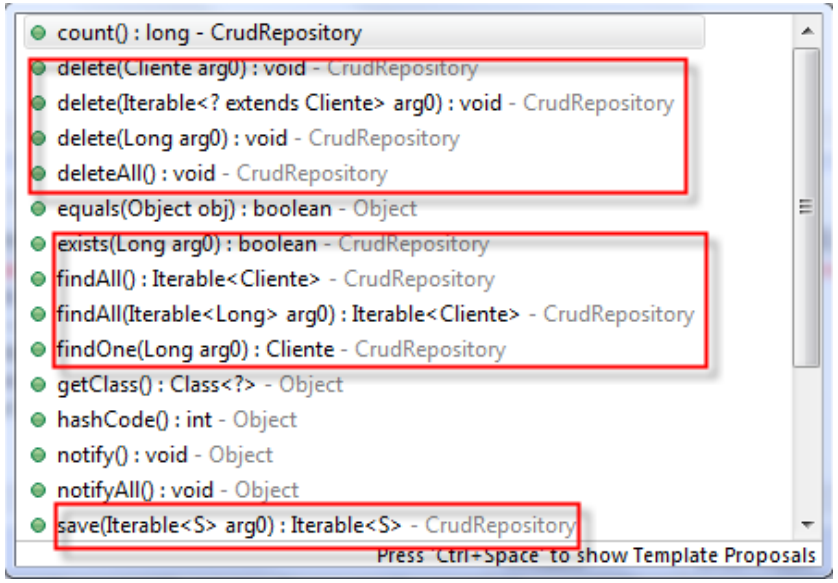


Spring Data es un proyecto de SpringSource para proporcionar un acceso fácil y unificado a diferentes clases de almacenamiento, tanto para bases de datos relacionales como almacenamientos NoSQL.

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)



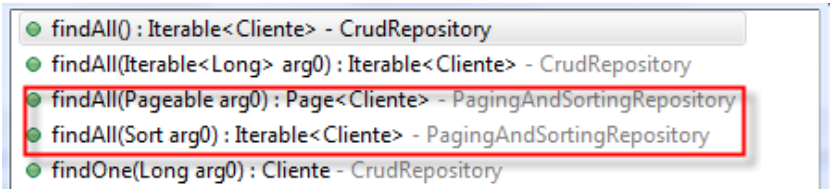
Extendiendo de CrudRepository obtenemos métodos autómaticos de

- 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



Extendiendo de PagingAndSortingRepository obtenemos métodos autómaticos 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 *JPATemplate* 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

```java
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
<?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>
```

RowMapper


```java
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;
    }

}
```

➢ 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

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

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

```xml
<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>
```

- **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);
```

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.

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 T> entities) Deletes the given entities.
void	**delete**(T entity) Deletes a given entity.
void	**deleteAll**() Deletes all entities managed by the repository.
void	**deleteInBatch**(Iterable<T> 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<T>	**findAll**() Returns all instances of the type.
Page<T>	**findAll**(Pageable pageable) Returns a Page of entities meeting the paging restriction provided in the Pageableobject.
List<T>	**findAll**(Sort sort) Returns all entities sorted by the given options.
T	**findOne**(ID id) Retrives an entity by its primary key.
void	**flush**() Flushes all pending changes to the database.
List<T>	**save**(Iterable<? extends T> entities) Saves all given entities.
T	**save**(T entity) Saves a given entity.
T	**saveAndFlush**(T entity) Saves an entity and flushes changes instantly.

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);
}
```

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
  - findByPropiedad1*And*Propiedad2…*And*PropiedadN
  - 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);

Queries JPQL

```
public interface UserRepository extends JpaRepository<User, Long> {
  @Query("select u from User u where u.emailAddress = ?1")
  User findByEmailAddress(String emailAddress);
}
```

@Query y @Param

```
public interface UserRepository extends JpaRepository<User, Long> {
  @Query("select u from User u
          where u.firstname = :firstname or u.lastname = :lastname")
  User findByLastnameOrFirstname(@Param("lastname") String lastname,
                                 @Param("firstname") String firstname);
}
```

Queries con nombre

@NamedQuery en entidad

```
@Entity
@NamedQuery(name = "User.findByEmailAddress",
  query = "select u from User u where u.emailAddress = ?1")
public class User {
}
```

<named-query/> en orm.xml de JPA

```
<named-query name="User.findByLastname">
  <query>select u from User u where u.lastname = ?1</query>
</named-query>
```

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

Actualizaciones en JPQL

@Modifying @Query

```
@Modifying
@Query("update User u set u.firstname = ?1 where u.lastname = ?2")
int setFixedFirstnameFor(String firstname, String lastname);
```

También se pueden usar

@NamedNativeQuery o <named-native-query />

para definir queries en el lenguaje SQL nativo de la BBDD, aunque de esta forma se perdería la característica de independencia de la plataforma.

Paginación y ordenación de consultas:

PageRequest, Sort y Page

```
@Test
public void pruebasPaginacion() throws Exception {

    Page<User> pageUsers = repo.findByLastname("Matthews",
            new PageRequest(0, 2));

    assertThat(pageUsers.getContent().size(), is(2));
    assertFalse(pageUsers.hasPreviousPage());

    Page<User> pageUsers = repo.findByLastname("Matthews“,
            new PageRequest(0, 10, new Sort(Direction.ASC, "lastname")));

    assertThat(pageUsers.getTotalElements(), is(2L));
    ...
}
```

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**()

Spring Data

Actividad 9

```xml
<?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>

</beans>
```

- Crear un proyecto Maven **spring-jdbc-test.**
- Configurar **DataSource** en el fichero **applicationContext.xml**.
- Crear una clase de Prueba que imprime en consola si la conexión esta cerrada y el nombre del catalogo.

```xml
<!-- AÑADIR EN POM.XML -->
<!-- spring-jdbc -->
<dependency>
    <groupId>org.springframework</groupId>
    <artifactId>spring-jdbc</artifactId>
    <version>5.0.4.RELEASE</version>
</dependency>
<!--  MySQL / MariaDB -->
<dependency>
    <groupId>mysql</groupId>
    <artifactId>mysql-connector-java</artifactId>
    <version>5.1.22</version>
</dependency>
```

```java
import java.sql.Connection;
import java.sql.SQLException;
import javax.sql.DataSource;
import org.springframework.context.ApplicationContext;
import org.springframework.context.support.ClassPathXmlApplicationContext;

public class Actividad9 {

    public static void main(String[] args) {
        ApplicationContext context = new ClassPathXmlApplicationContext("applicationContext.xml");

        try {
            DataSource ds = (DataSource) context.getBean("dataSource");
            Connection conn = ds.getConnection();
            System.out.println("¿Está cerrada la conexión? "+conn.isClosed() );
            System.out.println("BBDD conectada: "+conn.getCatalog() );
        }
        catch (SQLException ex) {
            ex.printStackTrace();
        }
    }
}
```

Spring Data

Actividad 10 – Sprint Data con JDBC

- Crear un bean **Beer**.
- Crear **BeerMapper** que transforma los datos del **ResultSet** al bean **Beer**.
- Crear una interfaz **BeerDAO** para definir las operaciones CRUD.
- Crear una clase **BeerDAOImpl** que realice las operaciones CRUD usando **JDBCTemplate** y que implemente la interfaz anterior.
- Crear una clase de prueba que ejecute los siguientes pasos:
  1. Imprimir el total de cervezas.
  2. Crear una nueva cerveza e insertarla en BD.
  3. Busca la cerveza insertada por su id.
  4. Actualiza ciertos datos de la cerveza encontrada.
  5. Elimina la cerveza por su id.

```java
public interface BeerDAO {
	public BeerVO findById(int id);
	public List<BeerVO> findAll();
	public int insert(BeerVO beer);
	public int update(BeerVO beer);
	public int delete(int id);
}
```

```java
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();
    }
}
```

```java
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();
    }
}
```

Querydsl. ¿Qué es?

Es un framework que permite la construcción de consultas SQL type-safe para múltiples backends incluyendo JPA, MongoDB y SQL en Java.

Los lenguajes de consultas (Sql en JDBC y HQL/JPA para Hibernate/JPA) presenta algunos problemas

- Es fácil introducir errores en las consultas.
- Hay acoplamiento entre la consulta a ejecutar y las columnas de la tabla a la que referencia.
- Este acoplamiento dificulta las tareas de mantenimiento.
  - Ejemplo: Cambio en una columna de una tabla obliga a redefinir la consulta.
- El proyecto Querydsl intenta solucionar este problema definiendo una API para la ejecución de consultas
- Esta API permite crear y utilizar consultas para una gran variedad de repositorios (JPA, Hibernate, JDO, JDBC nativo, Lucene, Hibernate, incluso bases de datos no-Sql (como MongoDB)

http://www.querydsl.com/

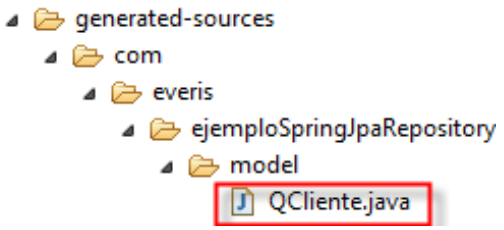
Spring Data

QueryDSL

Querydsl. ¿Cómo se utiliza?

- De manera automática cada clase del dominio genera automáticamente una clase Qnombre_entidad en la carpeta que se especifique en el plugin maven-apt-plugin .

```
@Entity
public class Cliente {

    @Id
    @GeneratedValue(strategy=GenerationType.AUTO)
    private long id;
    private String nombre;
    private String apellidos;
    private String nif;
```



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.

Recomendaciones:

- Se recomienda que las transacciones se definan a nivel de servicio de negocio, de forma que los DAO puedan usar los mismos paréntesis transaccionales, y que todo esto sea transparente para los controladores.
  - Controladores: sin definición de transacciones.
  - Servicios: propagation=REQUIRED, readOnly=false en los métodos con escritura

    propagation=SUPPORTS, readOnly=true el los métodos de sólo lectura
  - DAOs: JpaRepository por defecto declara los métodos implícitos con propagation=REQUIRED en general y readOnly=true para los métodos de sólo lectura.

- Otras estrategias configuran transacciones con propagation=REQUIRES_NEW en controladores y propagation=MANDATORY en servicios y DAOs, para asegurar que el paréntesis transaccional sea lo más amplio posible y que por errores en el desarrollo no se llamen a servicios ni a DAOs sin configurar transacciones.

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.

@Valid y BindingResult

```
@RequestMapping(“/grabar")
public @ResponseBody String grabar(
@Valid JavaBean bean, BindingResult result) {
     if (result.hasErrors()) {
          return “Hay errores de validación";
     } else {
          return "No hay errores";
     }
}
```

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 ofReadablePartial andReadableInstant.
@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 ofNumber.
@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 ofNumber.
@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 ofReadablePartial andReadableInstant.
@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-builtin

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, longand 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-builtin

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

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/actualizaones 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) { … }
```

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.

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"/>
```

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);
	}
}
```

Configurar @Transactional

Las transacciones admiten una serie de atributos:

- **rollbackFor**: clases que causarán ***rollback***
- **norollbackFor**
- **propagation**: propagación de la transacción, como en EJB.
- **timeout**: tiempo de espera

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

	@Transactional(
		rollbackFor=AltaPublicidadException.class,
		propagation=Propagation.REQUIRED
	)
	public void registrar(UsuarioVO user) {
		userDAO.registrarEnBD(user);
		userDAO.registrarEnListasDeCorreo(user);
	}
}
```

Spring Data

Spring Transaction Management

- Normalmente todo el código ejecutado dentro del alcance de una transacción será ejecutado en esa transacción.
- Spring ofrece opciones que especifican el comportamiento si un método es ejecutado cuando un contexto transaccional *ya existe* y si podemos continuar ejecutando el método en la transacción existente o podemos suspender la transacción y crear una nueva.
- Si un método es anotado como **@Transaccional** y se lanza una excepción que herede de *RuntimeException* se producirá un ***rollback***. En caso contrario, un ***commit***

➢ **PROPAGATION_REQUIRED**: Por defecto una transacción tiene una propagación de este tipo, si la transacción no existe la crea y si la tiene la aprovecha. Según esto, el primer método ***abre*** la transacción y el segundo la ***aprovecha*** por lo que todo está dentro de la misma transacción.

```
public class A {

    @Autowired
    private B b;

    @Transactional
    public void hazAlgoA() {
        b.hazAlgoB();
    }

}

public class B {

    @Transactional
    public void hazAlgoB() {
        hazAlgoTransaccionalmente();
    }
}
```

- **PROPAGATION_REQUIRES_NEW**: Abre una transacción nueva y pone en suspenso la anterior. Una vez el método marcado como REQUIRES_NEW termina se vuelve a la transacción anterior.

```
public class A {

    @Autowired
    private B b;

    @Transactional
    public void hazAlgoA() {
        b.hazAlgoB();
    }

}

public class B {

    @Transactional( propagation = Propagation.REQUIRES_NEW )
    public void hazAlgoB() {
        hazAlgoTransaccionalmente();
    }
}
```

Como el método de la clase B debe utilizar una nueva transacción, si algo falla en B, la transacción de A no hace ***Rollback***.

- **PROPAGATION_MANDATORY**: El método debe ser ejecutado dentro de una transacción, si ninguna transacción esta en progreso una excepción será lanzada.
- **PROPAGATION_NESTED**: El método debe ser ejecutado dentro de una transacción anidada si una transacción esta en progreso. Una transacción anidada se pueden ejecutar y revertir individualmente dentro de la transacción que la anida. Si no hay ninguna transacción que la anide, esta se comportara como PROPAGATION_REQUIRED.
- **PROPAGATION_NEVER**: El método actual no debe ejecutarse dentro de una transacción, si una transacción existente esta en progreso una excepción será lanzada.
- **PROPAGATION_NOT_SUPPORTED**: El método no debe ejecutarse dentro de una transacción, si existe otra en progreso, el método será suspendido hasta que esta ultima termine.
- **PROPAGATION_SUPPORTS**: Indica que el método actual no requiere un contexto de transacción, pero puede ser ejecutado dentro de una transacción si alguna esta en progreso. *Si existe transacción la aprovecha, sino no crea ninguna*.

everis (an **NTT DATA** Company)

Consulting, IT & Outsourcing Professional Services