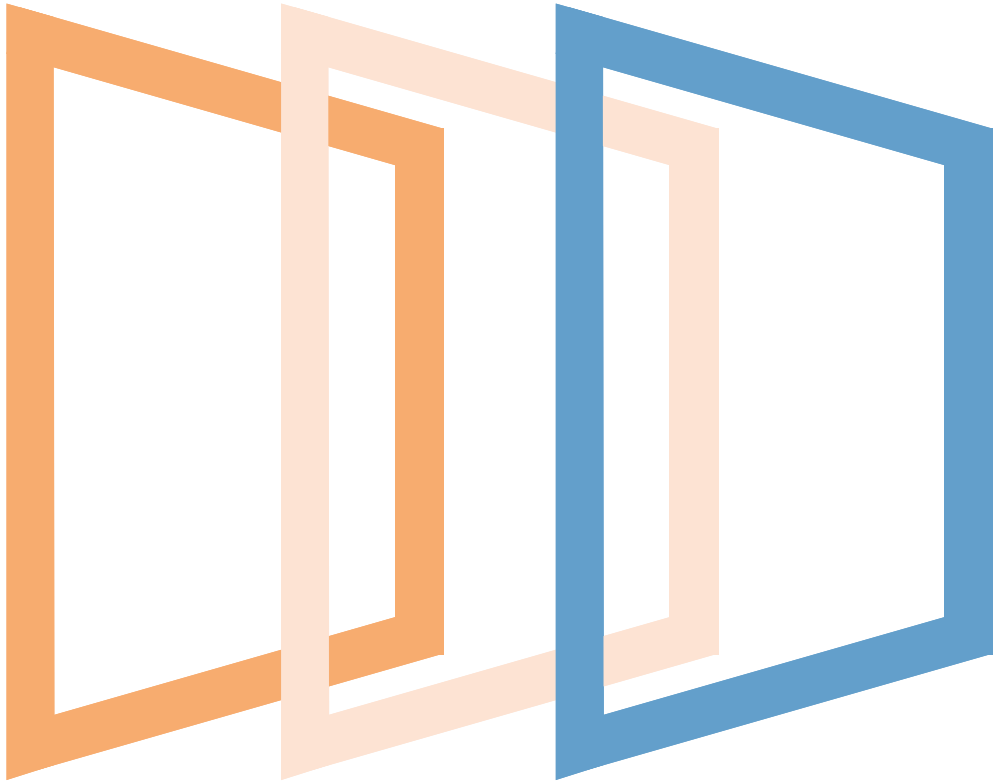
Microservicios con Spring



Junio 2019

minsait

An Indra company

Índice

01. Vista general y enlaces

02. Spring con Spring Boot

03. IoC con Spring Core

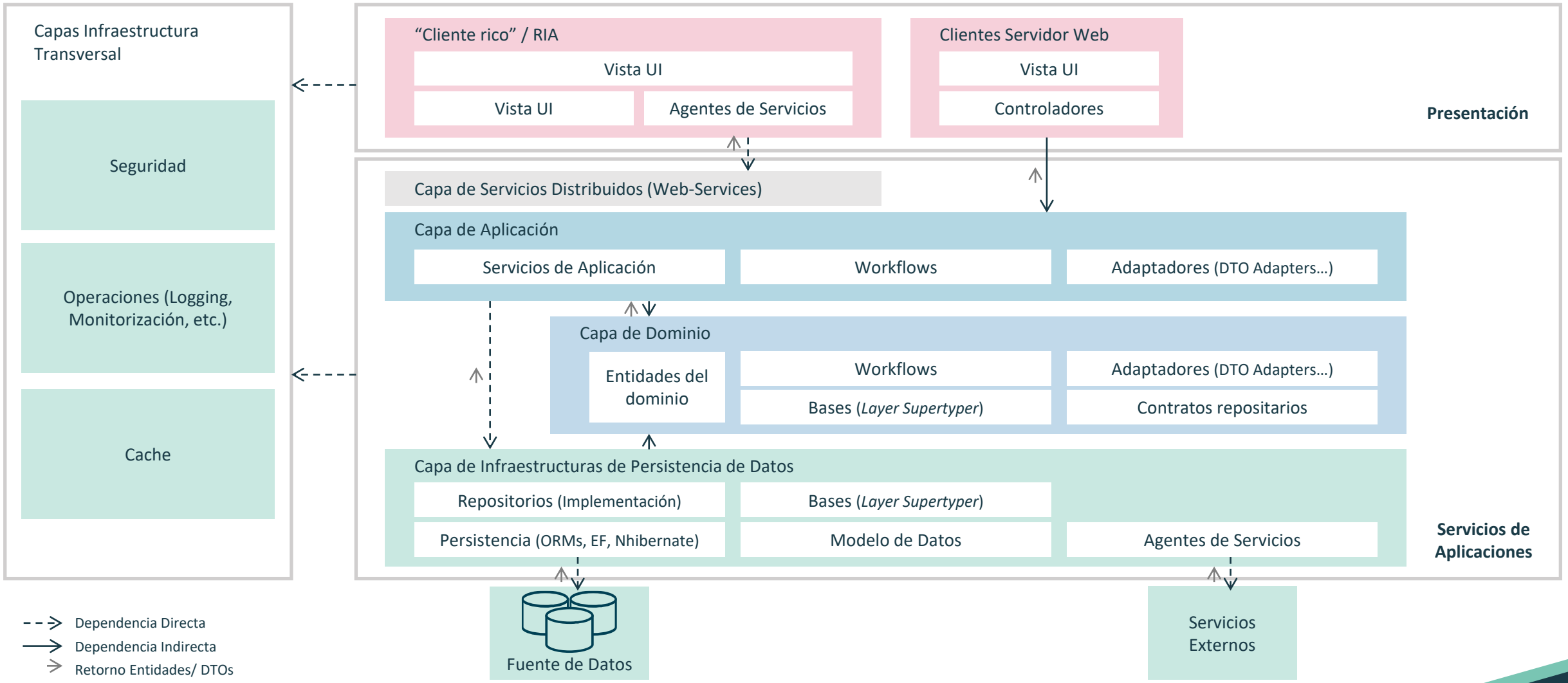
04. Acceso a datos con Spring Data

Vista general y enlaces

01

http://spring.io

Arquitectura N-Capas con Orientación al Dominio



Enlaces

Microservicios

https://martinfowler.com/articles/microservices.html

https://microservices.io/

Spring

https://spring.io/projects

Spring Core

https://docs.spring.io/spring-framework/docs/current/reference/html/

Spring Data

https://docs.spring.io/spring-data/jpa/docs/2.1.5.RELEASE/reference/html/

Spring MVC

https://docs.spring.io/spring/docs/current/spring-framework-reference/web.html

Spring HATEOAS

https://docs.spring.io/spring-hateoas/docs/0.25.1.RELEASE/reference/html

Spring Data REST

https://docs.spring.io/spring-data/rest/docs/3.1.5.RELEASE/reference/html/

Ejemplos:

https://github.com/spring-projects/spring-data-examples

https://github.com/spring-projects/spring-data-rest-webmvc

https://github.com/spring-projects/spring-hateoas-examples

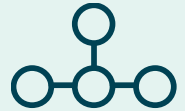
https://github.com/spring-projects/spring-integration-samples

Spring con Spring Boot

02

http://spring.io

Spring



- Inicialmente era un ejemplo hecho para el libro “J2EE design and development” de Rod Johnson en 2003, que defendía alternativas a la “visión oficial” de aplicación JavaEE basada en EJBs.
- Actualmente es un framework open source que facilita el desarrollo de aplicaciones java JEE & JSE (no está limitado a aplicaciones Web, ni a Java, pueden ser .NET, Silverlight, Windows Phone, etc.)
- Provee de un contenedor encargado de manejar el ciclo de vida de los objetos (beans) para que los desarrolladores se enfoquen a la lógica de negocio. Permite integración con diferentes frameworks.
- Surge como una alternativa a EJB’s
- Actualmente es un framework completo compuesto por múltiples módulos/proyectos que cubre todas las capas de la aplicación, con decenas de desarrolladores y miles de descargas al día
  - MVC.
  - Negocio (donde empezó originalmente).
  - Acceso a datos.

Características

- Ligero
  - No se refiere a la cantidad de clases sino al mínimo impacto que se tiene al integrar Spring.
- No intrusivo
  - Generalmente los objetos que se programan no tienen dependencias de clases específicas de Spring
- Flexible
  - Aunque Spring provee funcionalidad para manejar las diferentes capas de la aplicación (vista, lógica de negocio, acceso a datos) no es necesario usarlo para todo. Brinda la posibilidad de utilizarlo en la capa o capas que queramos.
- Multiplataforma
  - Escrito en Java, corre sobre JVM

Proyectos

Spring Framework

Da soporte básico para inyección de dependencia, gestión de transacciones, aplicaciones web, acceso a datos, mensajería y más.

Spring Data

Proporciona un enfoque coherente para el acceso a los datos: relacional, no relacional, map-reduce y más.

Spring Boot

Toma una visión por defecto de la creación de la aplicación Spring y lo pone en funcionamiento lo más rápido posible.

Spring Security

Protege tu aplicación con soporte de autenticación y autorización completo y extensible.

Spring Social

Conecta fácilmente tus aplicaciones con API de terceros como Facebook, Twitter, Linkedin y más.

Spring Cloud Data Flow

Un modelo operativo y de programación nativo en la nube para microservicios de datos componibles sobre una plataforma estructurada.

Spring Cloud

Proporciona un conjunto de herramientas para patrones comunes en sistemas distribuidos. Útil para construir e implementar microservicios.

Spring for Android

Proporciona componentes clave de Spring para usar en el desarrollo de aplicaciones de Android.

Módulos necesarios

Spring Framework

- Spring Core
  - Contenedor IoC (inversión de control) - inyector de dependencia.
- Spring MVC
  - Framework basado en MVC para aplicaciones web y servicios REST.

Spring Data

- Simplifica el acceso a los datos: JPA, bases de datos relacionales / NoSQL, nube.

Spring Boot

- Simplifica el desarrollo de Spring: inicio rápido con menos codificación.

Spring Boot

- Spring Boot es una herramienta que nace con la finalidad de simplificar aun más el desarrollo de aplicaciones basadas en el framework Spring Core: que el desarrollador solo se centre en el desarrollo de la solución, olvidándose por completo de la compleja configuración que actualmente tiene Spring Core para poder funcionar.
  - Configuración: Spring Boot cuenta con un complejo módulo que autoconfigura todos los aspectos de nuestra aplicación para poder simplemente ejecutar la aplicación, sin tener que definir absolutamente nada.
  - Resolución de dependencias: Con Spring Boot solo hay que determinar que tipo de proyecto estaremos utilizando, y Spring Boot se encarga de resolver todas las librerías/dependencias para que la aplicación funcione.
  - Despliegue: Spring Boot se puede ejecutar como una aplicación Stand-alone, pero también es posible ejecutar aplicaciones web, ya que es posible desplegar las aplicaciones mediante un servidor web integrado, como es el caso de Tomcat, Jetty o Undertow.
  - Métricas: Por defecto, Spring Boot cuenta con servicios que permite consultar el estado de salud de la aplicación, permitiendo saber si la aplicación está encendida o apagada, memoria utilizada y disponible, número y detalle de los Bean’s creado por la aplicación, controles para el prendido y apagado, etc.
  - Extensible: Spring Boot permite la creación de complementos, los cuales ayudan a que la comunidad de Software Libre cree nuevos módulos que faciliten aún más el desarrollo.
  - Productividad: Herramientas de productividad para desarrolladores como LiveReload y Auto Restart, funcionan en nuestro IDE favorito: Spring Tool Suite, IntelliJ IDEA y NetBeans.

Utilidades para empezar de cero. Entorno

Descargar Hibernate:

- http://hibernate.org/orm/downloads/

Descargar e instalar JDK:

- http://www.oracle.com/technetwork/java/javase/downloads/jdk8- downloads-2133151.html

Descargar y descomprimir Eclipse:

- https://www.eclipse.org/downloads/

Añadir a Eclipse las Hibernate Tools:

- Help > Eclipse Marketplace: JBoss Tools

Crear una User Library para Hibernate:

- Window > Preferences > Java > Build Path > User Libraries > New
- Add External JARs: \lib\required

Descargar y registrar la definición del driver JDBC:

- Window > Preferences > Data Management > Connectivity > Driver Definition > Add

Utilidades para empezar de cero. Instalación

https://spring.io/tools

Spring Tool Suite

- IDE gratuito, personalización del Eclipse.

Plug-in para Eclipse (VSCode, Atom)

- Help .Eclipse Marketplace …
  - Spring Tools 4 for Spring Boot.

Utilidades para empezar de cero. Crear proyecto

Desde web:

- https://start.spring.io/
- Descomprimir en el workspace
- Import--Maven --Existing Maven Project

Desde Eclipse:

- New Project --Sprint Boot --Spring Started Project

Dependencias:

- Web.
- JPA.
- JDBC (o proyecto personalizado).

Utilidades para empezar de cero. Dependencias opcionales

Serialización XML a cliente

```
<dependency>
<groupId>com.fasterxml.jackson.dataformat</group  Id>
    <artifactId>jackson-dataformat-xml</artifactId>
</dependency>
```

Utilidades para empezar de cero. Application

```
import org.springframework.boot.CommandLineRunner;  import
org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;

@SpringBootApplication
public class DemoApplication implements CommandLineRunner {

	public static void main(String[] args) {
		SpringApplication.run(ApiHrApplication.class, args);
	}

	@Override
	public void run(String... args) throws Exception {
		// Opcional: Procesar los args una vez arrancado SprintBoot

	}

}
```

Utilidades para empezar de cero. Configuración

@Configuration: Indica que esta es una clase usada para configurar el contenedor Spring.

@ComponentScan: Escanea los paquetes de nuestro proyecto en busca de los componentes que hayamos creado. Son las clases que utilizan las siguientes anotaciones: @Component, @Service, @Controller, @Repository.

@EnableAutoConfiguration: Habilita la configuración automática. Esta herramienta analiza el classpath y el archivo application.properties para configurar nuestra aplicación en base a las librerías y valores de configuración encontrados. Por ejemplo: al encontrar el motor de bases de datos H2, la aplicación se configura para utilizar este motor de datos. Al encontrar, Thymeleaf se crearan los beans necesarios para utilizar este motor de plantillas para generar las vistas de nuestra aplicación web.

@SpringBootApplication: Es el equivalente a utilizar las anotaciones:
@Configuration, @EnableAutoConfiguration y @ComponentScan

Utilidades para empezar de cero. Configuración

- Editar src/main/resources/application.properties:
  # Oracle settings
  spring.datasource.url=jdbc:oracle:thin:@localhost:1521:xe
  spring.datasource.username=hr
  spring.datasource.password=hr

  spring.datasource.driver-class=oracle.jdbc.driver.OracleDriver

  # MySQL settings
  spring.datasource.url=jdbc:mysql://localhost:3306/sakila spring.datasource.username=root
  spring.datasource.password=root
  spring.datasource.driver-class-name=com.mysql.cj.jdbc.Driver

  logging.pattern.console=%d{yyyy-MM-dd HH:mm:ss} %-5level %logger{36} - %msg%n logging.level.org.hibernate.SQL=debug

  server.port=8080

- Repetir con src/test/resources/application.properties

Utilidades para empezar de cero. Oracle Driver con Maven

- http://www.oracle.com/technetwork/database/features/jdbc/index-091264.html
- Instalación de Maven:
  - Descargar y descomprimir (https://maven.apache.org)
  - Añadir al PATH: C:\Program Files\apache-maven\bin
  - Comprobar en la consola de comandos: mvn -v
- Descargar el JDBC Driver de Oracle (ojdbc6.jar):
  - https://www.oracle.com/technetwork/apps-tech/jdbc-112010-090769.html
- Instalar el artefacto ojdbc en el repositorio local de Maven:
  - mvn install:install-file -Dfile=Path/to/your/ojdbc6.jar -DgroupId=com.oracle - DartifactId=ojdbc6 -Dversion=11.2.0 -Dpackaging=jar
- En el fichero pom.xml:

```
<dependency>
        <groupId>com.oracle</groupId>
        <artifactId>ojdbc6</artifactId>
        <version>11.2.0</version>
  </dependency>
```

Utilidades para empezar de cero. Proxy configuration: Maven

- Crear fichero setting.xml o editar %MAVEN_ROOT%/conf/setting.xml:

```
<?xml version="1.0" encoding="UTF-8"?>
<settings xmlns="http://maven.apache.org/SETTINGS/1.0.0"  xmlns:xsi="http://www.w3.org/2001/XMLSchema-
    instance"  xsi:schemaLocation="http://maven.apache.org/SETTINGS/1.0.0
    http://maven.apache.org/xsd/settings-1.0.0.xsd1.0.0.xsd">
  <localRepository>C:\directorio\local\.m2\repository</localRepository>
  <proxies>
      <proxy>
      <id>optional</id>
      <active>true</active>
      <protocol>http</protocol>
      <username>usuario</username>
      <password>contraseña</password>
      <host>proxy.dominion.com</host>
      <port>8080</port>
      <nonProxyHosts>local.net|some.host.com</nonProxyHosts>
    </proxy>
  </proxies>
</settings>
```

- Referenciarlo en Window → Preferences → Maven → User setting → User setting , browse…, seleccionar fichero recién creado, aceptar, update setting, aplicar y cerrar.

Utilidades para empezar de cero. Instalación de MySQL

- Descargar e instalar:
  - https://mariadb.org/download/
- Incluir en la seccion [mysqld] de

  %MYSQL_ROOT%/data/my.ini
  - default_time_zone='+01:00'
- Descargar bases de datos de ejemplos:
  - https://dev.mysql.com/doc/index-other.html
- Instalar bases de datos de ejemplos:
  - mysql -u root -p < employees.sql
  - mysql -u root -p < sakila-schema.sql
  - mysql -u root -p < sakila-data.sql

IoC con Spring Core

03

https://docs.spring.io/spring-framework/docs/current/reference/html/

Inversión de Control

- Inversión de control (Inversion of Control en inglés, IoC) es un concepto junto a unas técnicas de programación:
  - En las que el flujo de ejecución de un programa se invierte respecto a los métodos de programación tradicionales,
  - En los que la interacción se expresa de forma imperativa haciendo llamadas a procedimientos (procedure calls) o funciones.
- 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.

Inyección de Dependencias

- Inyección de Dependencias (en inglés Dependency Injection, DI) es un patrón de arquitectura orientado a objetos, en el que se inyectan objetos a una clase en lugar de ser la propia clase quien cree el objeto. Básicamente recomienda que las dependencias de una clase no sean creadas desde el propio objeto, sino que sean configuradas desde fuera de la clase.
- La inyección de dependencias (DI) procede del patrón de diseño más general que es la Inversión de Control (IoC).
- Al aplicar este patrón se consigue que las clases sean independientes unas de otras e incrementa la reutilización y la extensibilidad de la aplicación, además de facilitar las pruebas unitarias de las mismas.
- Desde el punto de vista de Java, un diseño basado en DI puede implementarse mediante el lenguaje estándar, dado que una clase puede leer las dependencias de otra clase por medio del API Reflection de Java y crear una instancia de dicha clase inyectándole sus dependencias.

Inyección

- La Inyección de Dependencias proporciona lo siguiente:
  - El código es más limpio.
  - El desacoplamiento es más eficaz, pues los objetos no deben de conocer donde están sus dependencias ni cuales son.
  - Facilidad en las pruebas unitaria e integración.

Introducción

- Spring proporciona un contenedor encargado de la inyección de dependencias (Spring Core Container).
- Este contenedor nos posibilita inyectar unos objetos sobre otros.
- Para ello, los objetos deberán ser simplemente JavaBeans.
- La inyección de dependencias será bien por constructor o bien por métodos setter.
- La configuración podrá realizarse bien por anotaciones Java o mediante un fichero XML (XMLBeanFactory).
- Para la gestión de los objetos tendrá la clase (BeanFactory).
- Todos los objetos serán creados como singletons si no se especifica lo contrario.

Modulo de dependencias

- Se crea el fichero de configuración applicationContext.xml y se guarda en el directorio src/META-INF.

```
<beans xmlns:aop="http://www.springframework.org/schema/aop"
       xmlns:context="http://www.springframework.org/schema/context"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xmlns="http://www.springframework.org/schema/beans"
       xsi:schemalocation="http://www.springframework.org/schema/beans
       http://www.springframework.org/schema/beans/spring-beans-2.5.xsd
       http://www.springframework.org/schema/aop  http://www.springframework.org/schema/aop/spring-aop-2.5.xsd  http://www.springframework.org/schema/context
       http://www.springframework.org/schema/context/spring-context-2.5.xsd">

       <context:component-scan base-package="es.miEspacio.ioc.services">

       </context:component-scan>

</beans>
```

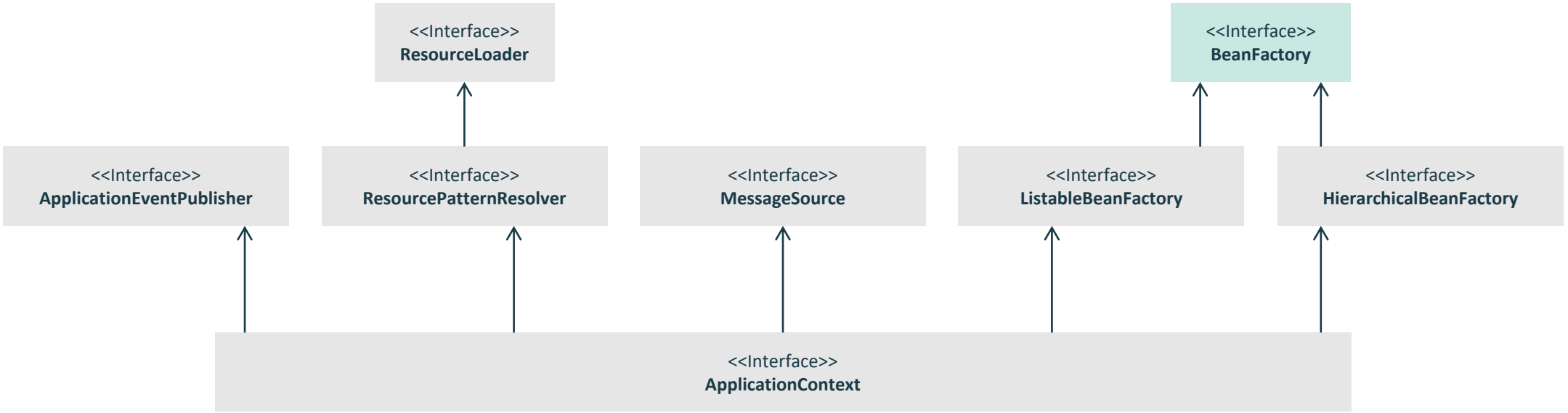
Beans

- Los beans se corresponden a los objetos reales que conforman la aplicación y que requieren ser inyectables: los objetos de la capa de servicio, los objetos de acceso a datos (DAO), los objetos de presentación (como las instancias Action de Struts), los objetos de infraestructura (como Hibernate SessionFactories, JMS Queues), etc.

```
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://www.springframework.org/schema/beans
        http://www.springframework.org/schema/beans/spring-beans.xsd">
    <!-- services -->
    <bean id="petStore" class="com.samples.PetStoreServiceImpl">
        <property name="accountDao" ref="accountDao"/>
        <property name="itemDao" ref="itemDao"/>
        <!-- additional collaborators and configuration for this bean go here -->
    </bean>
    <!-- more bean definitions for services go here -->
</beans>
```

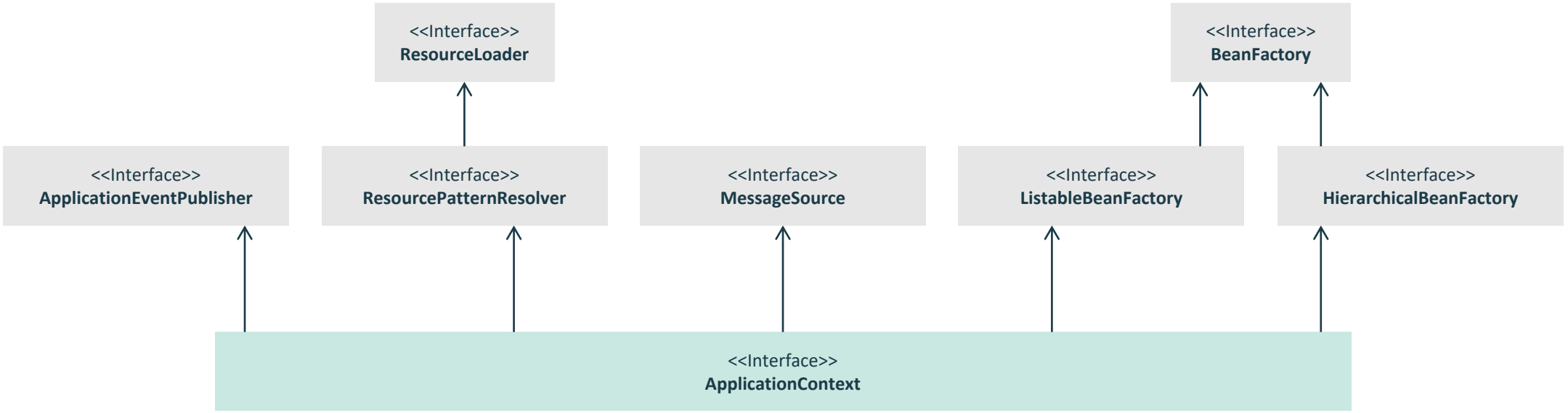
Bean factory

- Denominamos Bean Factory al contenedor Spring.
- Cualquier Bean Factory permite la configuración y la unión de objetos mediante la inyección de dependencia.
- Este Bean Factory también permite una gestión del ciclo de vida de los beans instanciados en él.
- Todos los contenedores Spring (Bean Factory) implementan el interface BeanFactory y algunos sub-interfaces para ampliar funcionalidades



Application Context

- Spring también soporta algo más avanzado, una “fábrica de beans” llamada contexto de aplicación.
- Application Context es una especificación de Bean Factory que implementa la interface ApplicationContext.
- En general, cualquier cosa que un Bean Factory puede hacer,
  - Un contexto de aplicación también lo puede hacer.



Uso de la inyección de dependencias

- Se crea un inyector partiendo de un módulo de dependencias.
- Se solicita al inyector las instancias para que resuelva las dependencias.

```
import org.springframework.beans.factory.BeanFactory;
import  org.springframework.context.ApplicationContext;
import org.springframework.context.support.ClassPathXmlApplicationContext;
public static void main(String[] args)  {
        ApplicationContext context = new ClassPathXmlApplicationContext("META-
        INF/applicationContext.xml");
        BeanFactory factory = context;
        Client client = (Client )factory.getBean("ID_Cliente");
        client.go();
}
```

- Muestra:
  - Este es un servicio…

Anotaciones IoC

Autodescubrimiento

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

Personalización

- @Configuration
- @Bean

Inyección

- @Autowire (@Inject)
- @Qualifier (@Named)
- @Value
- @PropertySource
- @Required
- @Resource

Otras

- @PostConstruct
- @PreDestroy

Estereotipos

- Spring define un conjunto de anotaciones core que categorizan cada uno de los componentes asociándoles una responsabilidad concreta.
  - @Component: Es el estereotipo general y permite anotar un bean para que Spring lo considere uno de sus objetos.
  - @Repository: Es el estereotipo que se encarga de dar de alta un bean para que implemente el patrón repositorio que es el encargado de almacenar datos en una base de datos o repositorio de información que se necesite. Al marcar el bean con esta anotación Spring aporta servicios transversales como conversión de tipos de excepciones.
  - @Service: Este estereotipo se encarga de gestionar las operaciones de negocio más importantes a nivel de la aplicación y aglutina llamadas a varios repositorios de forma simultánea. Su tarea fundamental es la de agregador.
  - @Controller: El último de los estereotipos que es el que realiza las tareas de controlador y gestión de la comunicación entre el usuario y el aplicativo. Para ello, se apoya habitualmente en algún motor de plantillas o librería de etiquetas que facilitan la creación de páginas.
  - @RestController: Es una especialización de controller que contiene las anotaciones @Controller y @ResponseBody (escribe directamente en el cuerpo de la respuesta en lugar de la vista).

Alcance

- Un aspecto importante del ciclo de vida de los Beans es si el contenedor creara una única instancia o tantas como ámbitos sean necesarios.
  - prototype: No reutiliza instancias, genera siempre una nueva.
- instancia. @Scope("prototype")
  - singleton: (Predeterminado) Instancia única para todo el contenedor Spring IoC. @Scope("singleton") @Singleton
  - Adicionalmente, en el contexto de un Spring Web ApplicationContext: @RequestScope @SessionScope @ApplicationScope

    request: Instancia única para el ciclo de vida de una sola solicitud HTTP. Cada solicitud HTTP tiene su propia instancia única.

    session: Instancia única para el ciclo de vida de cada HTTP Session.

    application: Instancia única para el ciclo de vida de un ServletContext.

    websocket: Instancia única para el ciclo de vida de un WebSocket.

Inyección

- La inyección se realiza con la anotación @Autowire:
  - En atributos:

    ```
    @Autowire
    private MyBeans myBeans;
    ```

  - En propiedades (setter):

    ```
    @Autowire
    public void setMyBeans(MyBeans value) { … }
    ```

  - En constructores
- Por defecto la inyección es obligatoria. Se puede marcar como opcional en cuyo caso, si no encuentra el Bean, inyectará un null.

  ```
  @Autowire(required=false) private MyBeans myBeans;
  ```

- Se puede completar @Autowire con la anotación @Lazy para inyectar un proxy de resolución lenta.

Acceso a ficheros de propiedades

- Localización (fichero .properties, .yml, .xml):
  - Por defecto: src/main/resources/application.properties
  - En la carpeta de recursos src/main/resources:

    @PropertySource("classpath:my.properties")
  - En un fichero local:

    @PropertySource("file://c:/cng/my.properties")
  - En una URL:

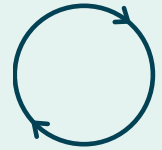
    @PropertySource("http://myserver/application.properties")
- Acceso directo:

  @Value("${spring.datasource.username}") private String name;
- Acceso a través del entorno:

  @Autowired private Environment env;

  env.getProperty("spring.datasource.username")

Ciclo de Vida



- Con la inyección el proceso de creación y destrucción de las instancias de los beans es administrado por el contenedor.
- Para poder intervenir en el ciclo para controlar la creación y destrucción de las instancias se puede:
  - Implementar las interfaces InitializingBean y DisposableBean de devoluciones de llamada.
  - Sobrescribir los métodos init() y destroy().
  - Anotar los métodos con @PostConstruct y @PreDestroy.
- Se pueden combinar estos mecanismos para controlar un bean dado.

Configuración por código

- Hay que crear una (o varias) clase anotada con @Configuration que contendrá un método por cada clase/interfaz (sin estereotipo) que se quiera tratar como un Bean inyectable.
- El método irá anotado con @Bean . Se debería llamar como la clase, en notación CamelCase, y devolver del tipo de la clase la instancia ya creada. Adicionalmente se puede anotar con @Scope y con @Qualifier.
  - public class MyBean { … }

    @Configuration
  - public class MyConfig {

    @Bean

    @Scope("prototype")
  - public MyBean myBean() { …]

Doble herencia

- Se crea el interfaz con la funcionalidad deseada:

```
public interface Service {
        public void go();
}
```

- Se implementa la interfaz en una clase (por convención, se usa el sufijo Impl):

```
import org.springframework.stereotype.Service;
@Service
@Singleton
public class ServiceImpl implements Service {
        public void go() {
                System.out.println("Este es un servicio...");
        }
}
```

Cliente

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


@Service("ID_Cliente")
public class Client {
    private final Service service;


    @Autowired
    public void setService(Service service){
            this.service = service;
    }

    public void go(){
            service.go();
    }
}
```

@Autowired establece que deben resolverse los parámetros mediante DI.

Anotaciones estándar JSR-330

- A partir de Spring 3.0, Spring ofrece soporte para las anotaciones estándar JSR-330 (inyección de dependencia). Esas anotaciones se escanean de la misma manera que las anotaciones de Spring.
- Cuando trabaje con anotaciones estándar, hay que tener en cuenta que algunas características importantes no están disponibles.

Anotaciones Spring	Anotaciones Estándar (javax.inject.*) JSR-330
@Autowired	@Inject
@Component	@Named / @ManagedBean
@Scope("singleton")	@Singleton
@Qualifier	@Qualifier / @Named
@Value	-
@Required	-
@Lazy	-

Acceso a datos con Spring Data

04

Spring Data

- Spring Framework ya proporcionaba soporte para JDBC, Hibernate, JPA o JDO, simplificando la implementación de la capa de acceso a datos, unificando la configuración y creando una jerarquía de excepciones común para todas ellas.

- Spring Data es un proyecto (subproyectos) de SpringSource cuyo propósito es unificar y facilitar el acceso a distintos tipos de tecnologías de persistencia, tanto a bases de datos relacionales como a las del tipo NoSQL.

- Spring Data viene a cubrir el soporte necesario para distintas tecnologías de bases de datos NoSQL integrándolas con las tecnologías de acceso a datos tradicionales, simplificando el trabajo a la hora de crear las implementaciones concretas.

- Con cada tipo de tecnología de persistencia, los DAOs (Data Access Objects) ofrecen las funcionalidades típicas de CRUD para objetos de dominio propios, métodos de búsqueda, ordenación y paginación. Spring Data proporciona interfaces genéricas para estos aspectos (CrudRepository, PagingAndSortingRepository) e implementaciones específicas para cada tipo de tecnología de persistencia.

Modelos: Entidades

- Una entidad es un tipo de clase dedicada a representar un modelo de dominio persistente que:
  - Debe ser publica (no puede ser estar anidada ni finalo tener miembros finales).
  - Deben tener un constructor público sin ningún tipode argumentos.
  - Para cada propiedad que queramos persistir debe haber un método get/set asociado.
  - Debe tener una clave primaria.
  - Debería sobrescribir los métodos equals y hashCode.
  - Debería implementar el interfaz Serializable para utilizar de forma remota.

Anotaciones JPA

Anotación	Descripción
@Entity	▪ Se aplica a la clase. ▪ Indica que esta clase Java es una entidad a persistir.
@Table(name="Tabla")	▪ Se aplica a la clase e indica el nombre de la tabla de la base de datos donde se persistirá la clase. ▪ Es opcional si el nombre de la clase coincide con el de la tabla.
@Id	▪ Se aplica a una propiedad Java e indica que este atributo es la clave primaria.
@Column(name="Id")	▪ Se aplica a una propiedad Java e indica el nombre de la columna de la base de datos en la que se persistirá la propiedad. ▪ Es opcional si el nombre de la propiedad Java coincide con el de la columna de la base de datos.
@Column(…)	▪ name: su nombre. ▪ length: su longitud. ▪ precision: número total de dígitos. ▪ scale: número de digitos decimales. ▪ unique: restriccion valor único. ▪ nullable: restriccion valor obligatorio. ▪ insertable: es insertable. ▪ updatable: es modificable.
@Transient	▪ Se aplica a una propiedad Java e indica que este atributo no es persistente

Asociaciones

- Uno a uno (Unidireccional):
  - En la entidad fuerte se anota la propiedad con la referencia de la entidad.
  - @OneToOne(cascade=CascadeType.ALL):
    - Esta anotación indica la relación uno a uno de las 2 tablas.
  - @PrimaryKeyJoinColumn:
    - Indicamos que la relación entre las dos tablas se realiza mediante la clave primaria.
- Uno a uno (Bidireccional):
  - Las dos entidades cuentan con una propiedad con la referencia a la otra entidad.

Asociaciones

- Uno a Muchos:
  - En Uno:
    - Dispone de una propiedad de tipo colección que contiene las referencias de las entidades muchos:
      - List: Ordenada con repetidos.
      - Set: Desordenada sin repetidos.
    - @OneToMany(mappedBy="propEnMuchos",cascade= CascadeType.ALL)
      - mappedBy: contendrá el nombre de la propiedad en la entidad Muchos con la referencia a la entidad Uno.
    - @IndexColumn (name="idx")
      - Opcional. Nombre de la columna que la tabla Muchos para el orden dentro de la Lista.
  - En Muchos:
    - Dispone de una propiedad con la referencia de la entidad Uno.
    - @ManyToOne
      - Esta anotación indica la relación de Muchos a Uno.
    - @JoinColumn (name="idFK")
      - Indicaremos el nombre de la columna que en la tabla muchos contiene la clave ajena a la tabla Uno.

Asociaciones

- Muchos a muchos (Unidireccional):
  - Dispone de una propiedad de tipo colección que contiene las referencias de las entidades muchos.
  - @ManyToMany(cascade=CascadeType.ALL):
    - Esta anotación indica la relación muchos a muchos de las 2 tablas.
- Muchos a muchos (Bidireccional):
  - La segunda entidad también dispone de una propiedad de tipo colección que contiene las referencias de las entidades muchos.
  - @ManyToMany(mappedBy="propEnOtroMuchos"):
    - mappedBy: Propiedad con la colección en la otra entidad para preservar la sincronicidad entre ambos lados.

Cascada

- El atributo cascade se utiliza en los mapeos de las asociaciones para indicar cuando se debe propagar la acción en una instancia hacia la instancias relacionadas mediante la asociación.
- Enumeración de tipo CascadeType:
  - ALL = {PERSIST, MERGE, REMOVE, REFRESH, DETACH}
  - DETACH (Separar)
  - MERGE (Modificar)
  - PERSIST (Crear)
  - REFRESH (Releer)
  - REMOVE (Borrar)
  - NONE
- Acepta múltiples valores:
  - @OneToMany(mappedBy="profesor", cascade={CascadeType.PERSIST, CascadeType.MERGE})

Mapeo de Herencia

- Tabla por jerarquía de clases:
  - Padre:
    - @Table("Account")
    - @Inheritance(strategy = InheritanceType.SINGLE_TABLE)
    - @DiscriminatorColumn(name="PAYMENT_TYPE")
  - Hija:
    - @DiscriminatorValue(value = "Debit")
- Tabla por subclases:
  - Padre:
    - @Table("Account")
    - @Inheritance(strategy = InheritanceType.JOINED)
  - Hija:
    - @Table("DebitAccount")
    - @PrimaryKeyJoinColumn(name = "account_id")
- Tabla por clase concreta:
  - Padre:
    - @Inheritance(strategy = InheritanceType.TABLE_PER_CLASS)
  - Hija:
    - @Table("DebitAccount")

Repositorio

- Un repositorio es una clase que actúa de mediador entre el dominio de la aplicación y los datos que le dan persistencia.
- Su objetivo es abstraer y encapsular todos los accesos a la fuente de datos.
- Oculta completamente los detalles de implementación de la fuente de datos a sus clientes.
- El interfaz expuesto por el repositorio no cambia aunque cambie la implementación de la fuente de datos subyacente (diferentes esquemas de almacenamiento).
- Se crea un repositorio por cada entidad de dominio que ofrece los métodos CRUD (Create-Read-Update-Delete), de búsqueda, ordenación y paginación.

Repositorio

- Con el soporte de Spring Data, la tarea repetitiva de crear las implementaciones concretas de DAO para las entidades se simplifica porque solo vamos se necesita un interfaz que extiende uno de los siguientes interfaces:
  - CrudRepository<T,ID>
    - count(), delete(T entity), deleteAll(), deleteAll(Iterable<? extends T> entities), deleteById(ID id), existsById(ID id), findAll(), findAllById(Iterable<ID> ids), findById(ID id), save(S entity), saveAll(Iterable<S> entities)
  - PagingAndSortingRepository<T,ID>
    - findAll(Pageable pageable), findAll(Sort sort)
  - JpaRepository<T,ID>
    - deleteAllInBatch, deleteInBatch, flush, getOne, saveAll, saveAndFlush
- En el proceso de inyección Spring implementa la interfaz antes de inyectarla:
  - public interface ProfesorRepository extends JpaRepository<Profesor, Long> {}
  - @Autowired
  - private ProfesorRepository repository;

Repositorio

- El interfaz puede ser ampliado con nuevos métodos que serán implementados por Spring:
  - Derivando la consulta del nombre del método directamente.
  - Mediante el uso de una consulta definida manualmente.
- La implementación se realizará mediante la decodificación del nombre del método. Dispone de una sintaxis especifica para crear dichos nombres:
  - List<Profesor> findByNombreStartingWiths(String nombre);
  - List<Profesor> findByApellido1AndApellido2OrderByEdadDesc(String apellido1, String apellido2);
  - List<Profesor> findByTipoIn(Collection<Integer> tipos);
  - int deleteByEdadGreaterThan(int valor);

Repositorio

- Prefijo consulta derivada:
  - find (read, query, get), count, delete
- Opcionalmente, limitar los resultados de la consulta:
  - Distinct, TopNumFilas y FirstNumFilas
- Expresión de propiedad: ByPropiedad
  - Operador (Between, LessThan, GreaterThan, Like, …) por defecto equal.
  - Se pueden concatenar varias con And y Or.
  - Opcionalmente admite el indicador IgnoreCase y AllIgnoreCase.
- Opcionalmente, OrderByPropiedadAsc para ordenar,
  - Se puede sustituir Asc por Desc. Admite varias expresiones de ordenación.
- Parámetros:
  - Un parámetro por cada operador que requiera valor, y debe ser del tipo apropiado.
- Parámetros opcionales:
  - Pageable, Sort

Repositorio

Palabra clave	Muestra	Fragmento de JPQL
And	findByLastnameAndFirstname	… where x.lastname = ?1 and x.firstname = ?2
Or	findByLastnameOrFirstname	… where x.lastname = ?1 or x.firstname = ?2
Is,Equals	findByFirstname, findByFirstna meIs, findByFirstnameEquals	… where x.firstname = ?1
Between	findByStartDateBetween	… where x.startDate between ?1 and ?2
LessThan	findByAgeLessThan	… where x.age < ?1
LessThanEqual	findByAgeLessThanEqual	… where x.age <= ?1
GreaterThan	findByAgeGreaterThan	… where x.age > ?1
GreaterThanEqual	findByAgeGreaterThanEqual	… where x.age >= ?1

Repositorio

Palabra clave	Muestra	Fragmento de JPQL
After	findByStartDateAfter	… where x.startDate > ?1
Before	findByStartDateBefore	… where x.startDate < ?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
StartingWith	findByFirstnameStartingWith	… where x.firstname like ?1 (parámetro enlazado con % anexado)
EndingWith	findByFirstnameEndingWith	… where x.firstname like ?1 (parámetro enlazado con % antepuesto)

Repositorio

Palabra clave	Muestra	Fragmento de JPQL
Containing	findByFirstnameContaining	… where x.firstname like ?1 (parámetro enlazado entre %)
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> ages)	… where x.age not in ?1
True	findByActiveTrue()	… where x.active = true
False	findByActiveFalse()	… where x.active = false
IgnoreCase	findByFirstnameIgnoreCase	… where UPPER(x.firstame) = UPPER(?1)

Repositorio

- Valor de retorno de consultas síncronas:
  - Find, read, query, get:
    - List<Entidad>
    - Stream<Entidad>
    - Optional<T>
  - Count, delete:
    - Long
- Valor de retorno de consultas asíncronas (deben ir anotadas con @Async):
  - Future<Entidad>
  - CompletableFuture<Entidad>
  - ListenableFuture<Entidad>

Repositorio

- Mediante consultas JPQL:

```
@Query("from Professor p where p.age > 67")
List<Professor> findRetired();

@Modifying
@Query("delete from Professor p where p.age > 67")
List<Professor> deleteRetired();
```

- Mediante consultas SQL nativas:
  - @Query("select * from Professors p where p.age between ?1 and ?2", nativeQuery=true)
    List<Professor> findActive(int inicial, int final);

DTO

- Un objeto de transferencia de datos (DTO) es un objeto que define cómo se enviarán los datos a través de la red.
- Su finalidad es:
  - Desacoplar del nivel de servicio de la capa de base de datos.
  - Quitar las referencias circulares.
  - Ocultar determinadas propiedades que los clientes no deberían ver.
  - Omitir algunas de las propiedades con el fin de reducir el tamaño de la carga.
  - Eliminar el formato de grafos de objetos que contienen objetos anidados, para que sean más conveniente para los clientes.
  - Evitar el “exceso” y las vulnerabilidades por publicación.

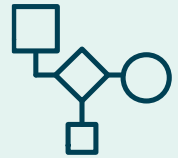
Lombok

- https://projectlombok.org/
- En las clases Java hay mucho código que se repite una y otra vez: constructores, equals, getters y setters. Métodos que quedan definidos una vez que dicha clase ha concretado sus propiedades, y que salvo ajustes menores, serán siempre sota, caballo y rey.
- Project Lombok es una biblioteca de Java que se conecta automáticamente al editor y crea herramientas que automatizan la escritura de Java.
- Mediante simples anotaciones ya nunca más vuelves a escribir otro método get o equals.

```
@Data @AllArgsConstructor @NoArgsConstructor public class MyDTO {

        private long id;

        private String name;

}
```

- La anotación @Value (no confundir con la de Spring) crea la versión de solo lectura.
- Es necesario agregar las bibliotecas al proyecto y configurar el entorno.

MapStruct



- http://mapstruct.org
- Las aplicaciones a menudo contienen modelos de objetos similares pero diferentes, donde los datos en dos modelos pueden ser similares pero la estructura y las responsabilidades de los modelos son diferentes. El mapeo de objetos facilita la conversión de un modelo a otro, permitiendo que los modelos separados permanezcan segregados.
- MapStruct facilita el mapeo de objetos, al determinar automáticamente cómo se mapea un modelo de objeto a otro, de acuerdo con las convenciones, de la misma forma que lo haría un ser humano, al tiempo que proporciona una API simple y segura de refactorización para manejar casos de uso específicos.

```
@Mappings({
  @Mapping(source = "itemId", target = "persistenceItemId"),
  @Mapping(target = "supplyType", ignore = true),
  @Mapping(target = "size", defaultValue = "10")
})
PersistenceMongoFindInventoryIDTO toPersistenceMongoFindInventoryIDTO(InventoryGetServiceIDTO
idto);
```

Proyecciones

- Los métodos de consulta de Spring Data generalmente devuelven una o varias instancias de la raíz agregada administrada por el repositorio. Sin embargo, a veces puede ser conveniente crear proyecciones basadas en ciertos atributos de esos tipos. Spring Data permite modelar tipos de retorno dedicados, para recuperar de forma más selectiva vistas parciales de los agregados administrados.
- La forma más sencilla de limitar el resultado de las consultas solo a los atributos deseados es declarar una interfaz o DTO que exponga los métodos de acceso para las propiedades a leer, que deben coincidir exactamente con las propiedades de la entidad:

```
public interface NamesOnly {
    String getNombre();
    String getApellidos();
}
```

- El motor de ejecución de consultas crea instancias de proxy de esa interfaz en tiempo de ejecución para cada elemento devuelto y reenvía las llamadas a los métodos expuestos al objeto de destino.

```
public interface ProfesorRepository extends JpaRepository<Profesor, Long> {
    List<NamesOnly> findByNombreStartingWith(String nombre);
}
```

Proyecciones

- Las proyecciones se pueden usar recursivamente.

```
interface PersonSummary {
    String getName();
    String getApellidos();
    DireccionSummary getDireccion();
interface
    DireccionSummary {  String
getCiudad();
    }
}
```

- En las proyecciones abiertas, los métodos de acceso en las interfaces de proyección también se pueden usar para calcular nuevos valores:

```
public interface NamesOnly {
    @Value("#{args[0] + ' ' + target.nombre + ' ' + target.apellidos}")  String
    getNombreCompleto(String tratamiento);
    default String getFullName() {
    return getNombre.concat(" ").concat(getApellidos());
    }
}
```

Proyecciones

- Se puede implementar una lógica personalizada mas compleja en un bean de Spring y luego invocarla desde la expresión SpEL:

```
@Component  class MyBean{
String getFullName(Person person) { … }
}
interface NamesOnly {  @Value("#{@myBean.getFullName(target)}")  String getFullName();
…
}
```

- Las proyecciones dinámicas permiten utilizar genéricos en la definición del repositorio para resolver el tipo de devuelto en el momento de la invocación:

```
public interface ProfesorRepository extends JpaRepository<Profesor, Long> {
<T> List<T> findByNombreStartingWith(String prefijo, Class<T> type);
}
dao.findByNombreStartingWith("J", ProfesorShortDTO.class)
```

Serialización Jackson

- Jackson es una librería de utilidad de Java que nos simplifica el trabajo de serializar (convertir un objeto Java en una cadena de texto con su representación JSON), y des-serializar (convertir una cadena de texto con una representación de JSON de un objeto en un objeto real de Java) objetos JSON.
- Jackson es bastante “inteligente” y sin decirle nada es capaz de serializar y des serializar bastante bien los objetos. Para ello usa básicamente la reflexión de manera que si en el objeto JSON tenemos un atributo “name”, para la serialización buscará un método “getName()” y para la des-serialización buscará un método “setName(String s)”.

```
ObjectMapper objectMapper = new ObjectMapper();

String jsonText = objectMapper.writeValueAsString(person);

Person person = new ObjectMapper().readValue(jsonText, Person.class);
```

- El proceso de serialización y des-serialización se puede controlar declarativamente mediante anotaciones:

  https://github.com/FasterXML/jackson-annotations

Serialización Jackson

- @JsonProperty: indica el nombre alternativo de la Propiedad en JSON.
  - @JsonProperty("name") public String getTheName() { ... } @JsonProperty("name") public void setTheName(String name) { ...
  - }
- @JsonFormat: especifica un formato para serializar los valores de fecha/hora.
  - @JsonFormat(shape = JsonFormat.Shape.STRING, pattern = "dd- MM-yyyy hh:mm:ss")
  - public Date eventDate;
- @JsonIgnore: marca que se ignore una propiedad (nivel miembro).
  - @JsonIgnore public int id;

Serialización Jackson

- @JsonIgnoreProperties: marca que se ignore una o varias propiedades (nivel clase).
  - @JsonIgnoreProperties({ "id", "ownerName" })
  - @JsonIgnoreProperties(ignoreUnknown=true) public class Item {
- @JsonInclude: se usa para incluir propiedades con valores vacíos, nulos, o predeterminados.
  - @JsonInclude(Include.NON_NULL) public class Item {
- @JsonAutoDetect: se usa para anular la semántica predeterminada de qué propiedades son visibles y cuáles no.
  - @JsonAutoDetect(fieldVisibility = Visibility.ANY) public class Item {

Serialización Jackson

- @JsonView: permite indicar la Vista en la que se incluirá la propiedad para la serialización / deserialización.

```
public class Views {
    public static class Partial {}
    public static class Complete extends Partial {}
}
public class Item {  @JsonView(Views.Partial.class)  public int id;
    @JsonView(Views.Partial.class) public String itemName;
    @JsonView(Views.Complete.class)  public String ownerName;
}
String result = new ObjectMapper().writerWithView(Views.Partial.class)
    .writeValueAsString(item);
```

Serialización Jackson

- @JsonFilter: indica el filtro que se utilizará durante la serialización (es obligatorio suministrarlo).

```
@JsonFilter("ItemFilter")  public class Item {
public int id;
public String itemName;  public String ownerName;
}
FilterProvider filters = new SimpleFilterProvider().addFilter("ItemFilter",  SimpleBeanPropertyFilter.filterOutAllExcept("id",  "itemName"));
MappingJacksonValue mapping = new  MappingJacksonValue(dao.findAll());
mapping.setFilters(filters);  return mapping;
```

Serialización Jackson

- @JsonManagedReference y @JsonBackReference: se utilizan para manejar las relaciones maestro/detalle marcando la colección en el maestro y la propiedad inversa en el detalle (multiples relaciones requieren asignar nombres unicos).
  - @JsonManagedReference public User owner;
    @JsonBackReference
  - public List<Item> userItems;

- @JsonIdentityInfo: indica la identidad del objeto para evitar problemas de recursión infinita.
  - @JsonIdentityInfo(
  - generator = ObjectIdGenerators.PropertyGenerator.class,
  - property = "id")
  - public class Item { public int id;

Serialización XML (JAXB)

- JAXB (Java XML API Binding) proporciona a una manera rápida, conveniente de crear enlaces bidireccionales entre los documentos XML y los objetos Java. Dado un esquema que especifica la estructura de los datos XML, el compilador JAXB genera un conjunto de clases de Java que contienen todo el código para analizar los documentos XML basados en el esquema. Una aplicación que utilice las clases generadas puede construir un árbol de objetos Java que representa un documento XML, manipular el contenido del árbol, y regenerar los documentos del árbol, todo ello en XML sin requerir que el desarrollador escriba código de análisis y de proceso complejo.
- Los principales beneficios de usar JAXB son:
  - Usa tecnología Java y XML.
  - Garantiza datos válidos.
  - Es rápida y fácil de usar.
  - Puede restringir datos.
  - Es personalizable.
  - Es extensible.

Anotaciones principales (JAXB)

- Para indicar a los formateadores JAXB como transformar un objeto Java a XML y viceversa se puede anotar (javax.xml.bind.annotation) la clases JavaBean para que JAXP infiera el esquema de unión.
- Las principales anotaciones son:
  - @XmlRootElement(namespace = "namespace"): Define la raíz del XML.
  - @XmlElement(name = "newName"): Define el elemento de XML que se va a usar.
  - @XmlAttribute(required=true): Serializa la propiedad como un atributo del elemento.
  - @XmlID: Mapea un propiedad JavaBean como un XML ID.
  - @XmlType(propOrder = { "field2", "field1",.. }): Permite definir en que orden se van escribir los elementos dentro del XML.
  - @XmlElementWrapper: Envuelve en un elemento los elementos de una colección.
  - @XmlTransient: La propiedad no se serializa.

Validaciones

- Desde la versión 3, Spring ha simplificado y potenciado en gran medida la validación de datos, gracias a la adopción de la especificación JSR 303. Este API permite validar los datos de manera declarativa, con el uso de anotaciones. Esto nos facilita la validación de los datos enviados antes de llegar al controlador REST.
- Las anotaciones se pueden establecer a nivel de clase, atributo y parámetro de método.
- Se puede exigir la validez mediante la anotación @Valid en el elemento a validar.
  - Public ResponseEntity<Object> create(@Valid @RequestBody Persona item)
- Para realizar la validación manualmente:

```
@Autowired
private Validator validator;
Set<ConstraintViolation<@Valid Persona>> constraintViolations = validator.validate( persona );
Set<ConstraintViolation<@Valid Persona>> constraintViolations = validator.validateProperty (persona, "nombre" );
```

Validaciones

- @Null: Comprueba que el valor anotado es null.
- @NotNull: Comprueba que el valor anotado no sea null.
- @NotEmpty: Comprueba si el elemento anotado no es nulo ni está vacío.
- @NotBlank: Comprueba que la secuencia de caracteres anotados no sea nula y que la longitud recortada sea mayor que 0. La diferencia con @NotEmptyes que esta restricción solo se puede aplicar en secuencias de caracteres y que los espacios en blanco finales se ignoran.
- @AssertFalse: Comprueba que el elemento anotado es falso.
- @AssertTrue: Comprueba que el elemento anotado es verdadero.

Validaciones

@Max(value=): Comprueba si el valor anotado es menor o igual que el máximo especificado.

@Min(value=): Comprueba si el valor anotado es mayor o igual que el mínimo especificado.

@Negative: Comprueba si el elemento es estrictamente negativo. Los valores cero se consideran inválidos.

@NegativeOrZero: Comprueba si el elemento es negativo o cero.

@Positive: Comprueba si el elemento es estrictamente positivo. Los valores cero se consideran inválidos.

@PositiveOrZero: Comprueba si el elemento es positivo o cero.

@DecimalMax(value=, inclusive=): Comprueba si el valor númerico anotado es menor que el máximo especificado, cuando inclusive= falso. De lo contrario, si el valor es menor o igual al máximo especificado.

@DecimalMin(value=, inclusive=): Comprueba si el valor anotado es mayor que el mínimo especificado, cuando inclusive= falso. De lo contrario, si el valor es mayor o igual al mínimo especificado.

Validaciones

@Past: Comprueba si la fecha anotada está en el pasado.

@PastOrPresent: Comprueba si la fecha anotada está en el pasado o en el presente.

@Future: Comprueba si la fecha anotada está en el futuro.

@FutureOrPresent : Comprueba si la fecha anotada está en el presente o en el futuro.

@Email: Comprueba si la secuencia de caracteres especificada es una dirección de correo electrónico válida.

@Pattern(regex=, flags=): Comprueba si la cadena anotada coincide con la expresión regular regex, considerando la bandera dada.

@Size(min=, max=): Comprueba si el tamaño del elemento anotado está entre min y max (inclusive).

Transacciones

- Por defecto, los métodos CRUD en las instancias del repositorio son transaccionales. Para las operaciones de lectura, el indicador readOnly de configuración de transacción se establece en true para optimizar el proceso. Todos los demás se configuran con un plano @Transactional para que se aplique la configuración de transacción predeterminada.
- Cuando se van a realizar varias llamadas al repositorio o a varios repositorios se puede anotar con @Transactional el método para que todas las operaciones se encuentren dentro de la misma transacción.
  - @Transactional
  - public void create(Pago pago) { … }
- Para que los métodos de consulta sean transaccionales:
  - @Override @Transactional(readOnly = false) public List<User> findAll();

Servicio

- Los servicios representan operaciones, acciones o actividades que no pertenecen conceptualmente a ningún objeto de dominio concreto. Los servicios no tienen ni estado propio ni un significado más allá que la acción que los definen. Se anotan con @Service.
- Podemos dividir los servicios en tres tipos diferentes:
  - Domain services
    Son responsables del comportamiento más específico del dominio, es decir, realizan acciones que no dependen de la aplicación concreta que estemos desarrollando, sino que pertenecen a la parte más interna del dominio y que podrían tener sentido en otras aplicaciones pertenecientes al mismo dominio.
  - Application services
    Son responsables del flujo principal de la aplicación, es decir, son los casos de uso de nuestra aplicación. Son la parte visible al exterior del dominio de nuestro sistema, por lo que son el punto de entrada-salida para interactuar con la funcionalidad interna del dominio. Su función es coordinar entidades, value objects, domain services e infrastructure services para llevar a cabo una acción.
  - Infrastructure services
    Declaran comportamiento que no pertenece realmente al dominio de la aplicación pero que debemos ser capaces de realizar como parte de este.

¡Gracias!

Presentación:
Arquitectura DSP
arquitecturaDSP@minsait.com

Avda. de Bruselas 35
28108 Alcobendas,
Madrid España

T +34 91 480 50 00
F +34 91 480 50 80
www.minsait.com

minsait

An Indra company

Mark Making the way forward

An Indra company