Spring

Victor Herrero Cazurro

Contenidos

¿Que es Spring?

Framework para el de desarrollo de aplicaciones java.

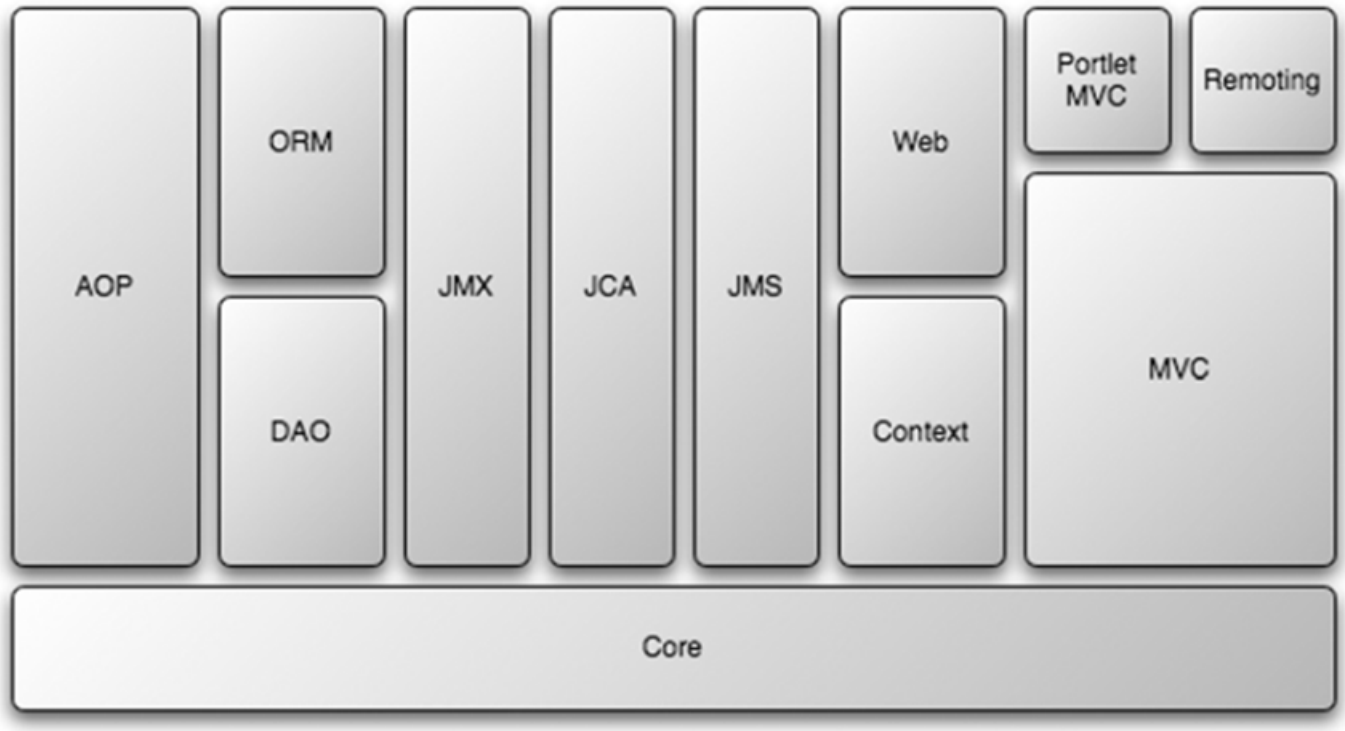
Es un contenedor ligero de POJOS que se encarga de la creación de beans (Factoría de Beans) mediante la Inversión de control y la inyección de dependencias.

Creado en 2002 por Rod Johnson

Spring se centra en proporcionar mecanismos de gestión de los objetos de negocio.

Spring es un framework idóneo para proyectos creados desde cero y orientados a pruebas unitarias, ya que permite independizar todos los componentes que forman a arquitectura de la aplicación.

Esta estructurado en capas, puede introducirse en proyectos de forma gradual, usando las capas que nos interesen, permaneciendo toda la arquitectura consistente.



IoC

Patrón de diseño que permite quitar la resposabilidad a los objetos de crear aquellos otros objetos que necesitan para llevar a cabo una acción, delegandola en otro componente, denominado Contedor o Contexto.

Los objetos simplemente ofrecen una determinada lógica y es el Contenedor que el orquesta esas logicas para montar la aplicación.

Inyección de dependencias

Patrón de diseño que permite desacoplar dos algortimos que tienen una relación de necesidad, uno necesita de otro para realizar su trabajo.

Se basa en la definición de Interfaces que definan que son capaces de hacer los objetos, pero no como realizan dicho trabajo (implementación).

```
interface GestorPersonas {

    void alta(Persona persona);

}

interface PersonaDao {

    void insertar(Persona persona);

}
```

Y la definición de propiedades en los objetos, que representarán la necesidad, la dependencia, de tipo la Interface antes creada, asociado a un método de **Set**, inyeccion por setter y/o a un constructor, inyección por construcción, para proporcionar la capacidad de que un elemeno externo, el contenedor, inyecte la dependencia al objeto.

```
class GestorPersonasImpl {

    private PersonaDao dao;

    //Inyeccion por contruccion
    public GestorPersonasImpl(PersonaDao dao){
        this.dao = dao;
    }

    //Inyeccion por setter
    public setDao(PersonaDao dao){
        this.dao = dao;
    }

    public void alta(Persona persona){
        dao.insertar(persona);
    }

}
```

Spring Context

Introduccion

El contexto de Spring describe una Factoria de Beans y la relación entre dichos Beans.

Se define con un objeto de tipo **ApplicationContext**.

Proporciona:

- Factoría de beans, previamente configurados. Realizando la inyección de dependencias.
- Manejo de textos con **MessageSource**, para internacionalización con i18n.
- Acceso a recursos, como URLs y ficheros.
- Propagación de eventos, para las beans que implementen **ApplicationListener**.
- Carga de múltiples contextos en jerarquía, permitiendo enfocar cada uno en cada capa.

ApplicationContext

ApplicationContext es la interfaz que modela el contexto de Spring, es el contenedor de todos los Beans que gestina Spring, de la cual se proporcionan diferentes implementaciones:

- **ClassPathXmlApplicationContext** → Carga el archivo de configuración desde un archivo XML que se encuentra en el classpath

```
ApplicationContext ctx = new ClassPathXmlApplicationContext("appContext.xml");
```

- **FileSystemXmlApplicationContext** → Carga el archivo de configuración desde un archivo en el sistema de ficheros

```
ApplicationContext ctx = new FileSystemXmlApplicationContext("appContext.xml");
```

- **AnnotationConfigApplicationContext** → Carga de archivos de configuracion anotados con **@Configuration**

```
AnnotationConfigApplicationContext context = new AnnotationConfigApplicationContext
(Configuracion.class);
```

NOTE

Para el caso de JavaConfig, se pueden registrar nuevos ficheros de configuración en caliente con

```
AnnotationConfigApplicationContext context = new
AnnotationConfigApplicationContext();
context.register(Configuracion.class);
context.refresh();
```

- **XmlWebApplicationContext** → Carga el archivo de configuración desde un XML contenido dentro

de una aplicación web

```
<context-param>
    <param-name>contextConfigLocation</param-name>
    <param-value>
        /WEB-INF/daoContext.xml
        /WEB-INF/applicationContext.xml
    </param-value>
</context-param>
<listener>
    <listener-class>
        org.springframework.web.context.ContextLoaderListener
    </listener-class>
</listener>
```

Como se ve, el Contexto de Spring hace referencia a uno o varios ficheros de configuración, donde se definen los Beans que formaran dicho contexto, en general habra dos formas de definir estos ficheros de contexto

- Por XML: Definiendo un fichero XML con los Beans del contexto de Spring.

```
<?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:p="http://www.springframework.org/schema/p"
    xsi:schemaLocation="http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans.xsd">

    <!-- Inyección de dependencias por setter -->
    <bean id="plantilla" class="beans.Persona" abstract="true">
        <property name="edad" value="99" />
        <property name="altura" value="1.75" />
    </bean>

    <!-- Inyección de dependencias por constructor -->
    <bean id="pConstructor" class="beans.Persona">
        <constructor-arg index="0" value="Pepe"/>
        <constructor-arg index="1" value="2"/>
        <constructor-arg index="2" value="7.3" />
    </bean>
</beans>
```

- Por JavaConfig: Definiendo una o varias clases anotadas con **@Configuration**.

```
@Configuration
public class Configuracion {}
```

Declaracion de Beans

Dado que será el contexto de Spring quien instancie los Beans, hay que indicarle como lo ha de hacerlo, debido a las distintas naturalezas de las clases, esta construcción se puede realizar de diferentes modos, estando en Spring soportados los siguientes:

- Construcción por constructor basico (sin parametros):

```
<bean id="bean1" class="ejemplos.Bean"/>
```

- Contrucción por contructor con parametros:

```
<bean id="lucia" class="javabeans.Persona">
    <constructor-arg name="nombre" value="Otralucia" />
    <constructor-arg name="edad" value="31" />
    <constructor-arg name="pareja" ref="fernando" />
</bean>
```

- Seteo de propiedades posterior a la contrucción, pero efectiva desde el primer momento que el Bean esta disponible:

```
<bean id="fernando" class="javabeans.Persona">
    <property name="pareja" ref="lucia"/>
</bean>
```

- **Factoría estática**: Clase con un método estatico, en el ejemplo **createInstance** que retorna un objeto de si mismo.

```
<bean id="bean1" class="ejemplos.Bean" factory-method="createInstance"/>
```

- **Factoría instanciada**: Clase con un método que retorna un objeto de otro tipo.

```
<bean id="myFactoryBean" class="ejemplos.FactoriaDeBeans"/>
<bean id="bean2" factory-bean="myFactoryBean" factory-method="createInstance" class=
"ejemplos.Bean"/>
```

NOTE	En el caso de necesitar pasar parametros a los métodos de factoria, estos se pasarán con la etiqueta <**constructor-arg**>

Para la definición con JavaConfig, el proceso se simplifica enormemente dado que unicamente hay que codificar las sentencias necesarias en java dentro de métodos anotados con @Bean en la clase de Configuracion, que retornen el Bean que se desea incluir en el contexto.

```
@Bean
public Persona juan(){
    return new Persona("Juan");
}
```

Para la definición con anotaciones, se emplearán alguna de las siguientes

- @Component → Beans generales
- @Repository → Beans de la capa de persistencia
- @Controller → Beans de la capa de controlador en MVC
- @Service → Beans de la capa de servicio
- @Named (JSR-330)

Para que estas anotaciones se puedan interpretar, habra que activarlas en el contexto, para ello:

- Si el contexto se define con xml

```
<context:component-scan base-package="com.curso.spring"></context:component-scan>
```

- Si el contexto se define con JavaConfig, en alguna de las clases de configuracion añadir la anotación @ConponentScan

```
@Configuration
@ComponentScan(basePackages={ "controllers" })
```

FactoryBean

Spring ofrece una interface **FactoryBean**, que permite definir factorias de Beans delegadas, es decir en vez de ser Spring el que instancia con la configuración, es la Factoria, instanciandose esta por Spring.

```
public interface FactoryBean<T> {
    T getObject() throws Exception;
    Class<T> getObjectType();
    boolean isSingleton();
}
```

Estas factorias unicamente han de proporcionar el objeto a construir, su tipo para que Spring sea capaz de discriminar a que factoria ha de pedir el objeto y si el objeto es unico, lo que permite a Spring cachearlo cuando se construye y no volver a emplear la factoria, este último escenario no es habitual.

Los APIs de Spring lo emplean ofreciendo algunas clases que implementan dicha interface como

- JndiFactoryBean
- LocalSessionFactoryBean
- LocalContainerEntityManagerFactoryBean

La particularidad de estas Factorias delegadas, es que al definir el API de Spring su forma, se puede emplear el identificador de la Factoria al hacer referencia a los Bean que crea, esto significa que definiendo un Bean de Spring de tipo Factoria de coches, con id **car**

```
<bean class = "a.b.c.MyCarFactoryBean" id = "car">
    <property name = "make" value ="Honda"/>
    <property name = "year" value ="1984"/>
</bean>
```

Se puede referenciar a los coches que fabrica con el id **car**

```
<bean class = "a.b.c.Person" id = "josh">
    <property name = "car" ref = "car"/>
</bean>
```

Ambitos (Scope)

El ambito de un Bean, representa el tiempo en ejecución en el cual el Bean esta disponible.

Se definen cuatro tipos de ambitos para una Bean

- **Singleton**: Una instancia única por JVM (por defecto), es decir el Bean esta diponible siempre.

```
<bean id="bean1" class="ejemplos.Bean"/>
```

- **Prototype**: Una nueva instancia cada vez que se pida el Bean. No tiene una vigencia establecida, depende de lo que la aplicación haga con el, pero para el contexto, una vez que lo contruye y lo proporciona, deja de existir.

```
<bean id="bean1" class="ejemplos.Bean" scope="prototype"/>
```

- **Request**: Ambito disponible en aplicaciones Web, permite asociar la vida de un Bean a una peticion que recibe el servidor, mientras que se pida al contexto el Bean dentro de la misma petición este siempre retornará el mismo Bean, al cambiar de petición, se dará otro.
- **Session**: Ambito disponible en aplicaciones Web, permite asociar la vida de un Bean a una sesión creada en el servidor para un usuario particular, mientras que se pida al contexto el Bean dentro de una petición asociada a una sesión establecida, este siempre retornará el mismo Bean, al cambiar de sesion, se dará otro.

Las beans singleton se instancian una vez que el contexto se carga. Esto se puede cambiar con la propiedad lazy-init="true" que indica al contexto que no cargue la bean hasta que se pida por primera vez.

Ciclo de Vida

El ciclo de vida de una bean en el contexto de Spring, permite conocer el momento de la creación y de la destrucción de la bean.

Se pueden definir métodos que se ejecuten en esos dos momentos empleando las propiedades **init-method** y **destroy-method**.

Si se mantiene una homegeneidad en los nombres de los métodos para todas las Bean, se puede definir en la etiqueta <beans/> los nombres de los métodos para facilitar la configuración con las propiedades **default-init-method** y **default-destroy-method**.

Existen algunas interfaces que nos permiten manejar el ciclo de vida sin necesidad de definir las propiedades init-method y destroy-method:

- InizializingBean → Obliga a la clase que la implemente a implementar el método afterPropertiesSet() que será llamado después de que todas las propiedades de la bean hayan sido configuradas.
- DisposableBean → Obliga a implementar el método destroy que será llamado justo antes de destruir la bean por el contenedor.

Extension del contexto

Se puede definir el contexto en multiples ficheros puediendose estos referenciar desde el principal.

Con XML

```
<import resource="cicloDeVida.xml"/>
```

Con Javaconfig

```
@Configuration
@Import(ConfiguracionPersistencia.class)
```

Tambien se pueden mezclar las dos formas de definir el contexto, importando desde la clase de configuración un fichero de XML

```
@ImportResource("classpath:/config/seguridad.xml")
```

O bien importando desde un XML una clase de configuracion, para lo unico que hay que hacer es escanear el paquete donde este dicha clase

```
<context:component-scan base-package="com.curso.spring.configuracion" />
```

NOTE	En las clases de configuración, se puede emplea la anotacion @Autowired para obtener la referencia a un Bean que este definido en otro fichero

Inyeccion de Dependencias

Las dependencias pueden ser de tipos complejos, por lo que se hará por referencia

```
<bean id="fernando" class="javabeans.Persona">
    <property name="pareja" ref="lucia"/>
</bean>
```

O de tipos simples, incluido String, que se hará por valor.

```
<bean id="fernando" class="javabeans.Persona">
    <property name="nombre" value="Fernando"/>
</bean>
```

Se pueden inyectar los Beans por Setter

```
<bean id="fernando" class="javabeans.Persona">
    <property name="pareja" ref="lucia"/>
</bean>
```

O por contructor

```
<bean id="lucia" class="javabeans.Persona">
    <constructor-arg name="nombre" value="Otralucia" />
    <constructor-arg name="edad" value="31" />
    <constructor-arg name="pareja" ref="fernando" />
</bean>
```

Tambien se pueden inyectar Beans exclusivos para otro Bean, a los que solo este tiene acceso, suelen ser anonimos

```
<bean id="fernando" class="javabeans.Persona">
    <property name="pareja">
        <bean class="javabeans.Persona">
            <constructor-arg name="nombre" value="Otralucia" />
            <constructor-arg name="edad" value="31" />
            <constructor-arg name="pareja" ref="fernando" />
        </bean>
    </property>
</bean>
```

Se puede indicar de forma explicita que se quiere inyectar un valor NULL con la etiqueta <null/>

```
<bean id="fernando" class="javabeans.Persona">
    <property name="pareja"><null/></property>
</bean>
```

Definición de Colecciones

Se dispone de etiquetas particulares para la definición de colecciones

- List

```
<list>
    <value>a list element followed by a reference</value>
    <ref bean="myDataSource" />
</list>
```

- Set

```
<set>
    <value>just some string</value>
    <ref bean="myDataSource" />
</set>
```

- Map

```
<map>
    <entry key=“JUAN” value=“Un valor”/>
    <entry key=“PEPE” value-ref=“miPersona” />
</map>
```

- Properties

```
<props>
    <prop key="adm"> administrator@somecompany.org</prop>
</props>
```

Ficheros de propiedades

Se pueden obtener literales de ficheros de propiedades a traves de expresiones EL (SpEl), para ello hay que cargar los ficheros properties como **PropertyPlaceholderConfigurer**

Desde el XML definiendo un bean

```
<bean class="org.springframework.beans.factory.config.PropertyPlaceholderConfigurer">
    <property name="locations">
        <array>
            <value>configuracion/configuracionBD.properties</value>
            <value>configuracion/configuracionServicio.properties</value>
        </array>
    </property>
</bean>
```

O de forma mas sencilla empleando el espacio de nombres **context**

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

O desde JavaConfig definiendo un bean

```
@Bean
public PropertyPlaceholderConfigurer propertyPlaceholderConfigurer() {

    PropertyPlaceholderConfigurer propertyPlaceholderConfigurer = new
PropertyPlaceholderConfigurer();
    propertyPlaceholderConfigurer.setLocation(new ClassPathResource("db.properties"));

    return propertyPlaceholderConfigurer;
}
```

O empleando la anotacion **PropertySource**

```
@Configuration
@PropertySource("classpath:db.properties")
public class ApplicationConfiguration {}
```

Y posteriormente se pueden refereciar las properties desde el XML

```
<bean class="com.ejemplo.DataSource">
    <property name="password" value="${service.provincias.password}"/>
    <property name="url" value="${service.provincias.url}"/>
    <property name="user" value="${service.provincias.user}"/>
</bean>
```

o desde las clases con la anotacion **@Value**, que puede afectar tanto a propiedades de una clase, como a parametros de un método.

```
@Value("${nombre}")
private String nombre;

@Bean
public DataSource dataSource(
        @Value("${dbDriver}") String driverClass,
        @Value("${dbUrl}") String jdbcUrl,
        @Value("${dbUsername}") String user,
        @Value("${dbPassword}") String password) {
}
```

NOTE	Notar que el acceso a las propiedades se hace con la sintaxis ${} y no #{}, esta última esta reservada a SpEL

Otra alternativa, es emplear la clase **Environment**

```
@Autowired
private Environment environment;
```

Leyendo posteriormente las propiedades

```
environment.getProperty("dbDriver")
```

Herencia

Se pueden reutilizar configuraciones de Beans a través de la herencia, para ello se dispone de dos propiedades

- **abstract** → Si es **true** indica que la bean declarada es abstracta y por tanto no podrá ser nunca instanciada, es decir no se le puede pedir al contenedor.

```
<bean id="personaGenerica" class="beans.Persona" abstract="true">
    <property name="nombre"><null /></property>
    <property name="cp" value="28001" />
</bean>
```

- **parent** → Indica el id de otra bean que se utilizará como padre. Concepto similar al extends en las clases Java, pero aplicado a los valores de las propiedades de los Beans.

```
<bean id="julio" parent="personaGenerica">
    <property name="nombre" value="Julio" />
</bean>
```

Autoinyección (Autowiring)

Consiste en la inyección de dependencias automática sin necesidad de indicarla en la configuración de una bean.

Se proporcionan 4 tipos de autowiring:

- Por nombre (byName) → El contenedor busca un bean cuyo nombre (ID) sea el mismo que el nombre de la propiedad. Si no se encuentra coincidencia la propiedad se devolverá sin dependencia.

```
<bean id="persona" class="beans.Persona" autowire="byName">
    <property name="nombre" value="persona"/>
</bean>
<bean id="direccion" class="beans.Direccion">
    <property name="cp" value="28900"/>
</bean>
```

- Por tipo (byType) → El contenedor busca un único bean cuyo tipo coincida con el tipo de la propiedad a inyectar. Si no se encuentra coincidencia la propiedad se devolverá sin dependencia y si se encuentra mas de una coincidencia el contenedor lanzará una excepción del tipo **org.springframework.beans.factory.UnsatisfiedDependencyException**

```
<bean id="dir" class="beans.Direccion">
    <property name="cp" value="28900"/>
</bean>
<bean id="persona" class="beans.Persona" autowire="byType">
    <property name="nombre" value="persona"/>
</bean>
```

- Por constructor (constructor) → El contenedor busca en los Beans disponibles en el contexto, unos que satisfagan los requerimientos del constructor del Bean en construccion, la resolución de las dependencias del constructor se realiza por tipo. Si no se encuentra coincidencia la propiedad se devolverá sin dependencia y si se encuentra mas de una coincidencia el contenedor lanzará una excepción del tipo **org.springframework.beans.factory.UnsatisfiedDependencyException**

```
<bean id="dir" class="beans.Direccion">
    <property name="cp" value="28900"/>
</bean>
<bean id="persona3" class="beans.Persona" autowire="constructor">
    <property name="nombre" value="persona"/>
</bean>
```

Para el ejemplo la clase Persona debe tener un constructor del tipo:

```
public Persona(Direccion dir){
    this.direccion= dir;
}
```

- Autodetectado (autodetect) → Se intenta realizar el autowiring por constructor. Si no se produce intentará realizarlo por tipo. Si no se encuentra coincidencia la propiedad se devolverá sin dependencia y si se encuentra mas de una coincidencia el contenedor lanzará una excepción del tipo **org.springframework.beans.factory.UnsatisfiedDependencyException**

```
<bean id="persona3" class="beans.Persona" autowire="autodetect">
    <property name="nombre" value="persona"/>
</bean>
```

Se puede configurar la autoinyección con anotaciones, empleando la anotacion @Autowired, aplicado al constructor

```
@Component
public class Negocio {

    private Persistencia persistencia;

    @Autowired
    public Negocio(Persistencia persistencia) {
        this.persistencia = persistencia;
    }
}
```

o al método de set.

```
@Component
public class Negocio {

    @Autowired
    private Persistencia persistencia;
}
```

Si no se requiere la inyección, es decir puede valer null, la anotación tiene una propiedad booleana **required**

```
@Component
public class Negocio {

    @Autowired(required=false)
    private Persistencia persistencia;
}
```

Tambien se puede emplear @Inject del estandar JSR-330

```
@Named
public class Negocio {

    @Inject
    private Persistencia persistencia;
}
```

Para el uso de las anotaciones, se ha de configurar el contexto para que sea capaz de interpretarlas, esto se consigue de dos formas con la etiqueta <context:component-scan> o con <context:annotation-config>

```
<context:annotation-config/>
```

La autoinyeccion si se define en el atributo o en el método de set, es por tipo, si se define en el constructor es por constructor y si se desea realizar una autoinyección por nombre, se dispone de @Qualifier, que asociado a la propiedad o al parametro del setter o el constructor, permite indicar el Id del bean a inyectar

```
@Component
public class Negocio {

    @Autowired
    @Qualifier("dao")
    private Persistencia persistencia;
}
```

Internacionalización

ApplicationContext extiende una interfaz llamada **MessageSource**, la cual proporciona funcionalidades para el manejo de properties dependientes del Locale.

Cuando se carga el contexto, automáticamente busca la bean MessageSource definida en la configuración, esta bean debe llamarse messageSource, si no se encuentra, se instancia un StaticMessageSource vacío.

Se proporcionan varias implementaciones de MessageSource

- ResourceBundleMessageSource.
- StaticMessageSource.
- ReloadableResourceBundleMessageSource.

El más usado es el primero, que permite definir las ubicaciones de los ficheros .properties que se van a

utilizar.

```
<bean id="messageSource" class=
"org.springframework.context.support.ResourceBundleMessageSource">
    <property name="basenames">
        <list>
            <value>format</value>
            <value>exceptions</value>
            <value>windows</value>
        </list>
    </property>
</bean>
```

En este caso, deberemos tener tres ficheros en el raíz del classpath, con los nombres format.properties, exceptions.properties, windows.properties. Admitiendo el resto de ficheros con los sufijos de locale para internacionalización.

La interface **MessageSource** proporciona los siguientes métodos para el manejo de los properties:

- String getMessage(String code, Object[] args, String default, Locale loc) → Método básico para recuperar un mensaje del MessageSource. Si no se encuentra un mensaje, se usa el default.
- String getMessage(String code, Object[] args, Locale loc) → similar pero sin mensaje por defecto.

```
String[] string = {"Alta"};
context.getMessage("com.ejemplo.cliente.alta.titulo", string, Locale.getDefault());
```

- String getMessage(MessageSourceResolvable resolvable, Locale locale) → En este caso, MessageSourceResolvable, agrupa los argumentos de los métodos anteriores.

Profiles

Permiten activar Beans definidos en el contexto.

Con XML basta con incluir la propiedad profile en la etiqueta <**beans**>

```
<beans profile="desarrollo">
    <bean id="servicio" class="con.curso.spring.ServicioImpl" />
</beans>
```

Con Javaconfig basta con incluir la anotacion **@Profile** en la clase Configuration.

```
@Configuration
@Profile("desarrollo")
public class Configuracion {}
```

o en el Bean que se quiera activar o desactivar.

```
@Configuration
public class Configuracion {
    @Bean
    @Profile("desarrollo")
    public Servicio servicio() {
        return new ServicioImpl();
    }
}
```

Una vez declarado los Beans pertenecientes al Profile, habrá que activar el Profile deseado, para ello se emplea una variable de entorno que puede ser definida de varias formas

- A traves del contexto de Spring

```
public static void main(String[] args) {
    AnnotationConfigApplicationContext context = new AnnotationConfigApplicationContext
(Configuracion.class);
    context.getEnvironment().setActiveProfiles("desarrollo");
    context.refresh();
}
```

- U obteniendo la referencia al objeto generado en el contexto que permite interaccionar con las varialbes de entorno **ConfigurableEnvironment**

```
@Configuration
public class Configuracion {
    @Autowired
    private ConfigurableEnvironment env;

    @PostConstruct
    public void init(){
        env.setActiveProfiles("someProfile");
    }
}
```

- Estableciendo directamente la variable de entorno programaticamente con el API de la JRE

```
public static void main(String[] args) {
      System.setProperty(AbstractEnvironment.ACTIVE_PROFILES_PROPERTY_NAME, "desarrollo"
);
      AnnotationConfigApplicationContext context = new
AnnotationConfigApplicationContext(Configuracion.class);
    }
}
```

- Estableciendo el parametro de la JVM al arrancarla

```
-Dspring.profiles.active=desarrollo
```

- Estableciendo la variable de entorno en el SO

```
export spring_profiles_active=dev
```

- Para aplicaciones Web, definiendo del el fichero web.xml un **context-param**

```
<context-param>
    <param-name>spring.profiles.active</param-name>
    <param-value>live</param-value>
</context-param>
```

- O para aplicaciones Web que no definan el web.xml

```
@Configuration
public class InicializadorWeb implements WebApplicationInitializer {

    @Override
    public void onStartup(ServletContext servletContext) throws ServletException {
        servletContext.setInitParameter("spring.profiles.active", "desarrollo");
    }
}
```

- En Test con la anotacion @ActiveProflies

```
@RunWith(SpringJUnit4ClassRunner.class)
@ContextConfiguration(classes = { Configuracion.class })
@ActiveProfiles("desarrollo")
public class ServicioTest {}
```

Eventos

En Spring se proporcionan una interface **ApplicationListener** y una clase **ApplicationEvent**, para el modelado de listener y eventos.

Hay tres eventos proporcionados por Spring:

- **ContextRefreshedEvent** → Evento publicado cuando el ApplicationContext se inicializa o refresca.
- **ContextClosedEvent** → Evento publicado cuando se cierra el ApplicationContext.
- **RequestHandledEvent** → Un evento específico para aplicaciones web que avisa que todas las beans de una petición HTTP han sido servida.

Pudiendose definir nuevos eventos extendiendo la clase **ApplicationEvent**.

```
public class MiEvento extends ApplicationEvent{
    public MiEvento(Object source) {
        super(source);
    }
}
```

Para definir Listener asociados a cada uno de dichos eventos se ha de definir un Bean de Spring sobre una clase que implemente la interface **ApplicationListener** indicando el tipo de evento que se escuchará.

```
public class Escuchador implements ApplicationListener<ContextRefreshedEvent>{
    public void onApplicationEvent(ContextRefreshedEvent event) {
        System.out.println("Se ha lanzado un evento de inicio: " + event.getSource());
    }
}
```

Cuando un escuchador de evento (ApplicationListener) recibe una notificación de un evento, se invoca su método **onApplicationEvent**.

Se pueden publicar eventos llamando al método **publishEvent()** de **ApplicationContext**, pasando como parámetro una instancia de una clase que implemente **ApplicationEvent**.

```
context.publishEvent(new MiEvento("origen"));
```

Tambien empleando el objeto de tipo **ApplicationEventPublisher** que se crea en el contexto de Spring

```
public class PublicadorDeEventos {

    @Autowired
    private ApplicationEventPublisher applicationEventPublisher;
}
```

O haciendo que un Bean implemente la interface **ApplicationEventPublisherAware**

```
public class PublicadorEventos implements ApplicationEventPublisherAware{

    private ApplicationEventPublisher applicationEventPublisher;

    public void setApplicationEventPublisher(ApplicationEventPublisher
applicationEventPublisher) {
        this.applicationEventPublisher = applicationEventPublisher;
    }
}
```

NOTE	Los Listener reciben el evento de forma síncrona, bloqueándo el hilo hasta terminar de procesar el evento.

Procesado Asincrono

Para realizar un procesado asincrono de eventos, habrá que reconfigurar el bean con id **applicationEventMulticaster** de tipo **SimpleApplicationEventMulticaster**, que define Spring por defecto, para ello simplemente hay que definir en el contexto de spring un bean con el mismo id.

```
<bean id="applicationEventMulticaster" class=
"org.springframework.context.event.SimpleApplicationEventMulticaster">
    <property name="taskExecutor" ref="simpleAsyncTaskExecutor"/>
</bean>
```

Este Bean tiene la posibilidad de definir un **taskExecutor**, que por defecto no viene configurado, que se encargará de lanzar los eventos en otros hilos.

```
<bean id="simpleAsyncTaskExecutor" class=
"org.springframework.core.task.SimpleAsyncTaskExecutor"></bean>
```

Una vez definido, el tratamiento de los eventos, se hará en hilo distintos.

Spring Expression Language (SpEL)

Las expresiones SpEL siguen el formato #{ } y no ${}, que se reserva para el acceso a los properties.

Lenguaje de expresiones con el que Spring permite realizar

- Referencias a Beans por Id.

```
#{servicio}
```

- Invocar metodos y acceder a propiedades.
- Definir espresiones matematicas, relacionales y logicas
- Busquedas por expresiones regulares
- Manipulacion de colecciones
- Acceso a propiedades del sistema

```
#{systemProperties['disc.title']}
```

Operadores

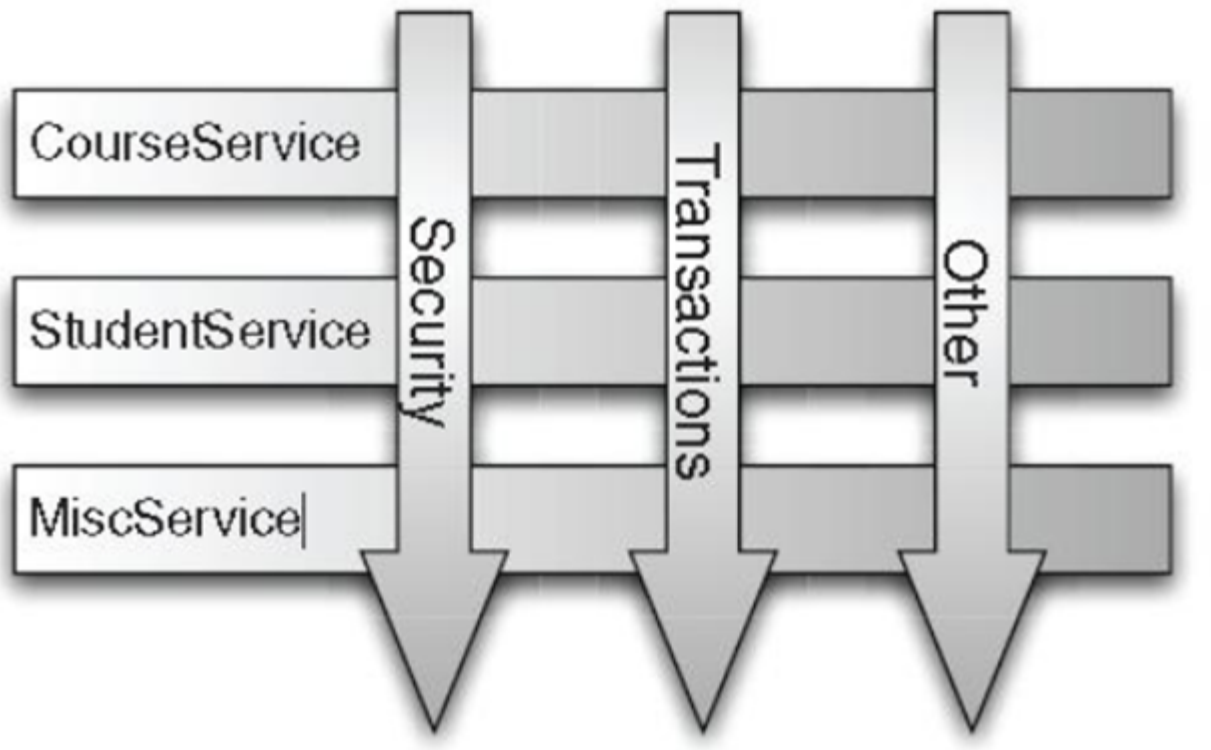
El operador T(), permite referenciar a clases con métodos estaticos para invocarlos

```
#{T(System).currentTimeMillis()}
```

Spring AOP

Framework que proporciona capacidades para definir beans que actuan como Aspectos.

Un Aspecto es una funcionalidad o tarea genérica que puede afectar a varias/muchas funcionalidades/clases, sin formar parte de la funcionalidad en sí, se les denomina servicios horizontales declarativos.



Algunos ejemplos son el logging, la gestión transaccional y la seguridad.

Definiciones importantes

- **Target Object**: Objeto observado por uno o más aspectos.
- **Proxy**: Objeto que suplanta al **Target Object** permitiendo la ejecución del **Aspecto**.
- **Joint Point**: Punto en la ejecución de un programa al que se asocia la ejecución de un aspecto en Spring todos los Joint Point son ejecuciones de métodos.
- **Advice**: Momento respecto al **Joint Point** en el que se ejecuta el Aspecto. Puede tomar valores como: around, before y after.
- **Point Cut**: Predicado que selecciona **Join Point**. Un **Advice** se asocia a una expresión **Point Cut** y se ejecuta en cualquier **Join Point** que coincide con el **Point Cut**.
- **Introduction**: **Aspecto** que permite declarar nuevos métodos o campos de un tipo.
- **Advisor**: Es un Aspecto, que define un único **Advice**.

Configuración de AOP con Spring

Se precisa incluir la dependencia opcional AspectJWeaver

```
<dependency>
    <groupId>org.aspectj</groupId>
    <artifactId>aspectjweaver</artifactId>
    <version>1.8.13</version>
</dependency>
```

La definición de AOP en Spring empleando configuración por XML, se basa en el espacio de nombres **aop**, teniendo que estar toda la configuración contenida en la etiqueta <**aop:config**>

```
<aop:config>
    <aop:pointcut
        expression="execution(* com.ejemplo.Negocio.*(com.ejemplo.Persona)) &amp;amp;amp;
args(p)"
        id="servicioPointCut"/>
    <aop:aspect ref="aspecto">
        <aop:after method="despuesDeServicio" pointcut-ref="servicioPointCut"
            arg-names="p"/>
        <aop:before method="antesDeServicio" pointcut-ref="servicioPointCut"
            arg-names="p"/>
        <aop:around method="durante" pointcut-ref="servicioPointCut"
            arg-names="p" />
    </aop:aspect>
    <aop:aspect>
        <aop:declare-parents
            types-matching="com.ejemplo.Negocio+"
            implement-interface="com.ejemplo.AmpliacionServicio"
            default-impl="com.ejemplo.AmpliacionServicioImpl" />
    </aop:aspect>
</aop:config>
```

De emplearse las anotaciones, se han de activar, en los XML con

```
<aop:aspectj-autoproxy/>
```

Y con Javaconfig

```
@Configuration
@EnableAspectJAutoProxy
public class Configuracion {

}
```

Pointcut

Predicado definido con el lenguaje de AspectJ, que selecciona métodos de Beans de Spring, que cuando se vayan a ejecutar provocarán la ejecución de un Aspecto.

```
<aop:pointcut expression="execution(* com.ejemplo.Negocio.*(com.ejemplo.Persona))
&amp;&amp; args(p)" id="servicioPointCut"/>
```

Las expresiones de AspectJ que definen el predicado, se estructuran así:

```
execution(modifiers-pattern? ret-type-pattern declaring-type-pattern? name-pattern(param-
pattern) throws-pattern?)
```

El siguiente ejemplo afectará a las llamadas de todos los métodos públicos de todas las clases del paquete **com.ats.test**, independientemente de los parametros que reciban o del tipo de datos retornados.

```
execution(public com.ats.test.*.*(..))
```

El * es un comodín para los tipos de acceso, nombres de clase y métodos, mientras que para los argumentos de un método es **..**

Cuando en la seleccion, se quiere hacer participe a los argumentos, se pone el tipo

```
execution(* transfer(int,int))
```

Se pueden combinar expresiones con los operadores &&, || y !, con el significado tradicional que tienen en java.

NOTE	En el caso de definición XML, debe usarse en vez de &&, **&&**.

Los **Point Cut** soportados por Spring son

- **execution** –> para seleccionar la ejecución de un método
- **within** –> para seleccionar la ejecución de cualquier metódo de un tipo especificado.

```
within(com.ats.test.*)
```

- **this** –> para seleccionar la ejecución de algún método de una bean cuyo **proxy** implemente la interfaz especificada.

```
this(com.ats.AccountService)
```

- **target** → para seleccionar ejecución de algún método de una bean cuyo **target object** implemente la interfaz especificada.

```
target(com.ats.services.MyService)
```

- **args** → seleccionar ejecución de algún método de un tipo cuyos argumentos son instancias de los tipos dados.

Aspect

Define el Bean de Spring que contiene la logica que se desea ejecutar y los **Advice** que definen cuando se ha de ejecutar cada uno de los métodos de Bean.

```
<aop:aspect ref="aspecto">
</aop:aspect>
```

NOTE	Cuando se define un **Aspecto**, se esta asociando un Bean, el Aspecto, con un método a suplantar seleccionado por un PointCut y un momento en el que se ha de ejecutar un método del Aspecto respecto a la ejecución del método interceptado (Advice).

Advice

Relación entre el método del **Aspecto** y el momento cuando se ha de ejecutar.

- **Before advice** → Se ejecuta antes del join point, pero no puede cortar el flujo de ejecución, a no ser que lance una excepción.

```
<aop:before method="antesDeServicio" pointcut-ref="servicioPointCut" arg-names="p"/>
```

```
@Before("execution(* com.ejemplo.Negocio.*(com.ejemplo.Persona)) &amp;&amp; args(p)")
public void antesDeServicio(Persona p){
    System.out.println("Se va a proceder a ejecutar el servicio con la persona: " + p);
}
```

- **After returning advice** → Se ejecuta cuando el joint point ha sido ejecutado de forma correcta.
- **After throwing advice** → Se ejecuta cuando el joint point termina lanzando una excep.
- **After (finally) advice** → Se ejecuta tanto al retornar correctamente, como cuando se lanza una

excep.

```
<aop:aspect ref="aspecto">
    <aop:after method="despuesDeServicio" pointcut-ref="servicioPointCut" arg-names="p"/>
</aop:aspect>
```

```
@After("execution(* com.ejemplo.Negocio.*(com.ejemplo.Persona)) &amp;&amp; args(p)")
public void despuesDeServicio(Persona p) {
    System.out.println("Se ha terminado de ejecutar el servicio con la persona: " + p);
}
```

- **Around advice** → Permite ejecutar acciones antes y después de un join point. Puede cortar la ejecución devolviendo el valor que considere o lanzando una excepción.

```
<aop:around method="durante" pointcut-ref="servicioPointCut" arg-names="p" />
```

```
@Around("execution(* com.ejemplo.Negocio.*(com.ejemplo.Persona)) &amp;&amp; args(p)")
public void durante(ProceedingJoinPoint joinPoint, Persona p){
    long inicio = System.currentTimeMillis();
    System.out.println("Antes de ejecutar el servicio con la persona " + p);
    p.setEdad(123);
    try {
        joinPoint.proceed();
    } catch (Throwable e) {
        e.printStackTrace();
    }
    long duracion = System.currentTimeMillis() - inicio;
    System.out.println("Despues de ejecutar el servicio, habiendo durado la operaci  n: "
+ duracion + " con la persona " + p);
}
```

Introduction

Define la extensión de un determinado tipo de Bean con los mñetodos que define la interface, empleando una implementación por defecto de dichos métodos.

```
<aop:aspect>
    <aop:declare-parents
        types-matching="com.ejemplo.Negocio+"
        implement-interface="com.ejemplo.AmpliacionServicio"
        default-impl="com.ejemplo.AmpliacionServicioImpl" />
</aop:aspect>
```

Advisor

Para la definición del Bean que represente el **Aspecto**, se emplearán las siguientes interfaces

- MethodBeforeAdvice
- ThrowsAdvice
- AfterReturningAdvice
- MethodInterceptor

```
public class AdvisorBefore implements MethodBeforeAdvice{
    public void before(Method method, Object[] args, Object target) throws Throwable {
        System.out.println("Antes de ejecutar el metodo");
    }
}
```

Para la definición del **Advisor**, se asociará el **Point Cut** con el Bean que implemente alguna de las interfaces anteriores.

```
<aop:advisor advice-ref="advisorBefore" pointcut="execution(*
com.ejemplo.Negocio.*(com.ejemplo.Persona))"/>
```

AOP con el API de Spring (ProxyFactoryBean)

Spring Jdbc

Framework que pretende dar soporte avanzado a la especificación JDBC de Java.

Se basa en la utilización de un Bean de tipo **JdbcTemplate**, que permite abstraer de la complejidad del Api de JDBC, ya que gestiona la conexión y el procesamiento de los **ResultSet**, para esto último ayudado de una implementación de **RowMapper<T>**.

Spring tambien ofrece una clase de soporte **JdbcDaoSupport** que embebe el uso de **JdbcTemplate**.

Para emplearlo, habra que definir una clase que extienda de **JdbcDaoSupport**

```
public class JdbcTemplateClienteDao extend JdbcDaoSupport implements ClienteDao {}
```

Una vez definida la clase, desde los métodos se tiene acceso a una instancia de **JdbcTemplate** con **getJdbcTemplate()**.

Y es esta clase **JdbcTemplate**, la que proporciona las funcionalidades para realizar las consultas, a traves de

- query()

```
List<CLiente> clientes = getJdbcTmplate().query("SELECT * FROM CLIENTE", new
ClienteRowMapper());
```

- queryForObject()
- update()

```
Map<String, Object> param = new HashMap<String, Object>();
        param.put("nombre", cliente.getNombre());
        param.put("direccion", cliente.getDireccion());
        param.put("telefono", cliente.getTelefono());
getTemplate().update("insert into clientes.cliente (nombre,direccion,telefono) values
(:nombre,:direccion,:telefono)", param);
```

Los métodos de la plantilla aceptarán de forma generica un String que represente la consulta a ejecutar, que se puede parametrizar con la sintaxis **:<nombre de parametro>** y un mapa, donde las claves son los nombres de los parametros de la consulta.

Siendo **RowMapper** una clase de utilleria que abstrae la extracción de los datos del **ResultSet**

```
public class RowMapperClienteImpl implements RowMapper<Cliente> {
    public Cliente mapRow(ResultSet rs, int rowNum) throws SQLException {
        return new Cliente(rs.getInt("id"),
                rs.getString("nombre"),
                rs.getString("direccion"),
                rs.getString("telefono")
                );
    }
}
```

Spring ORM

Framework que permite la integración de otros framework Orm como Hibernate, Jpa o MyIbatis.

De forma analoga a lo que sucede con JDBC, se proporcionan plantillas que encapsulan el manejo del API del ORM, facilitando la creación de la capa de persistencia, así se dispone de

- HibernateTemplate
- JpaTemplate

Estas plantillas dependerán de objetos del API correspondiente como son **SessionFactory** o **EntityManagerFactory**

Hibernate

Spring proporciona las siguientes implementaciones para construir los objetos **SessionFactory**.

- AnnotationSessionFactoryBean → Permite interpretar las anotaciones **@Entity**

```
<bean id="sessionFactory" class=
"org.springframework.orm.hibernate3.annotation.AnnotationSessionFactoryBean">
    <property name="dataSource" ref="dataSource"/>
    <property name="packagesToScan">
        <list>
            <value>com.curso.spring</value>
        </list>
    </property>
    <property name="hibernateProperties">
        <props>
            <propkey="dialect">org.hibernate.dialect.DerbyDialect</prop>
        </props>
    </property>
</bean>
```

En vez del **packageToScan**, se pueden indicar las **annotatedClases**

```
<bean id="sessionFactory" class=
"org.springframework.orm.hibernate3.annotation.AnnotationSessionFactoryBean">
    <property name="dataSource" ref="dataSource"/>
    <property name="annotatedClasses">
        <list>
            <value>com.entidades.Persona</value>
            <value>com.entidades.Factura</value>
        </list>
    </property>
    <propertyname="hibernateProperties">
        <props>
            <propkey="dialect">org.hibernate.dialect.DerbyDialect</prop>
        </props>
    </property>
</bean>
```

- LocalSessionFactoryBean

```
<bean id="sessionFactory" class=
"org.springframework.orm.hibernate3.LocalSessionFactoryBean">
    <property name="dataSource" ref="dataSource" />
    <property name="mappingResources">
        <list>
            <value>Persona.hbm.xml</value>
        </list>
    </property>
    <property name="hibernateProperties">
        <props>
            <prop key="dialect">org.hibernate.dialect.DerbyDialect</prop>
        </props>
    </property>
</bean>
```

Con **JavaConfig** la creación del Bean seria similar

```
@Bean
@Autowired
public LocalSessionFactoryBean sessionFactory(DataSource ds) {
    LocalSessionFactoryBean lsfb = new LocalSessionFactoryBean();
    //Entidades
    lsfb.setPackagesToScan("com.atsistemas.persistencia.core.entidades");
    //Origen de datos
    lsfb.setDataSource(ds);
    //Otras propiedades
    Properties hibernateProperties = lsfb.getHibernateProperties();

    hibernateProperties.setProperty("hibernate.dialect",
"org.hibernate.dialect.MySQL57Dialect");
    hibernateProperties.setProperty("hibernate.show_sql", "true");
    hibernateProperties.setProperty("hibernate.format_sql", "true");
    hibernateProperties.setProperty("hibernate.hbm2ddl.auto", "create");

    return lsfb;
}
```

A mayores del **Bean** de tipo **SessionFactory**, se precisa tambien de uno de tipo **TransactionManager**, que gestione la apertura y cierre de las transacciones, la declaracion de este Bean y su configuracion, esta explicado mas adelante en la seccion de **Transacciones**

Jpa

Spring proporciona las siguientes implementaciones para construir los objetos **EntityManager**.

- LocalContainerEntityManagerFactoryBean → Los Bean de entidad son gestionados por el contenedor, por lo que no es necesario definir el fichero **META-INF/persistence.xml** y en cambio si será necesario definir un **VendorAdapter** de entre los que proporciona Spring, dependiendo de la implementacion de JPA a emplear.
- EclipseLinkJpaVendorAdapter.
- HibernateJpaVendorAdapter.
- OpenJpaVendorAdapter.
- TopLinkJpaVendorAdapter.

```
<bean id="jpaVendorAdapter" class=
"org.springframework.orm.jpa.vendor.HibernateJpaVendorAdapter">
    <property name="database" value="Derby"/>
    <property name="showSql" value="true"/>
    <property name="generateDdl" value="false"/>
    <property name="databasePlatform" value="org.hibernate.dialect.DerbyDialect"/>
</bean>
```

Ademas de la definicion del Bean de Spring

```
<bean id="emf" class="org.springframework.orm.jpa.LocalContainerEntityManagerFactoryBean
">
    <property name="dataSource"ref="dataSource"/>
    <property name="jpaVendorAdapter" ref="jpaVendorAdapter"/>
    <property name="packagesToScan" value="com.cursospring.modelo.entidad"/>
    <property name="jpaProperties">
        <props>
            <prop key="hibernate.dialect">org.hibernate.dialect.DerbyDialect</prop>
            <prop key="hibernate.show_sql">true</prop>
            <prop key="hibernate.format_sql">true</prop>
            <!--<prop key="hibernate.current_session_context_class">thread</prop>--><!--
"jta", "thread" y "managed" -->
            <prop key="hibernate.hbm2ddl.auto">create</prop><!-- create, validate, update
-->
            <prop key="hibernate.default_schema">CLIENTES</prop>
        </props>
    </property>
</bean>
```

- LocalEntityManagerFactoryBean → Los Bean de entidad son gestionados por la aplicación, por lo que hay que definir el fichero **META-INF/persistence.xml**

```
<persistence xmlns=http://java.sun.com/xml/ns/persistence version="1.0">
    <persistence-unitname="miPersistenceUnit">
        <class>com.entidades.Persona</class>
        <properties></properties>
    </persistence-unit>
</persistence>
```

Y se define el Bean de Spring

```
<bean id="emf" class="org.springframework.orm.jpa.LocalEntityManagerFactoryBean">
    <property name="persistenceUnitName" value="miPersistenceUnit"/>
</bean>
```

Gestion de Excepciones

Se proporciona un API de excepciones mas extendido que el que proporciona JDBC, con su **SQLException**.

Hibernate, por ejemplo, también ofrece un conjunto de excepciones ampliado sobre JDBC, el problema, es que son solo útiles para este framework, si queremos independencia con el framework de persistencia, podemos emplear el API de excepciones de Spring.

Dado que muchas de las excepciones no son recuperables, Spring opta por ofrecer una jerarquía de excepciones **unchecked** (RuntimeException), es decir, ¿sino no se puede arreglar el problema porque esa necesidad añadir try o throws?

Para que Spring capture las excepciones lanzadas por hibernate y las convierta en excecpiones de Spring, hay que definir un nuevo bean el en contexto de Spring.

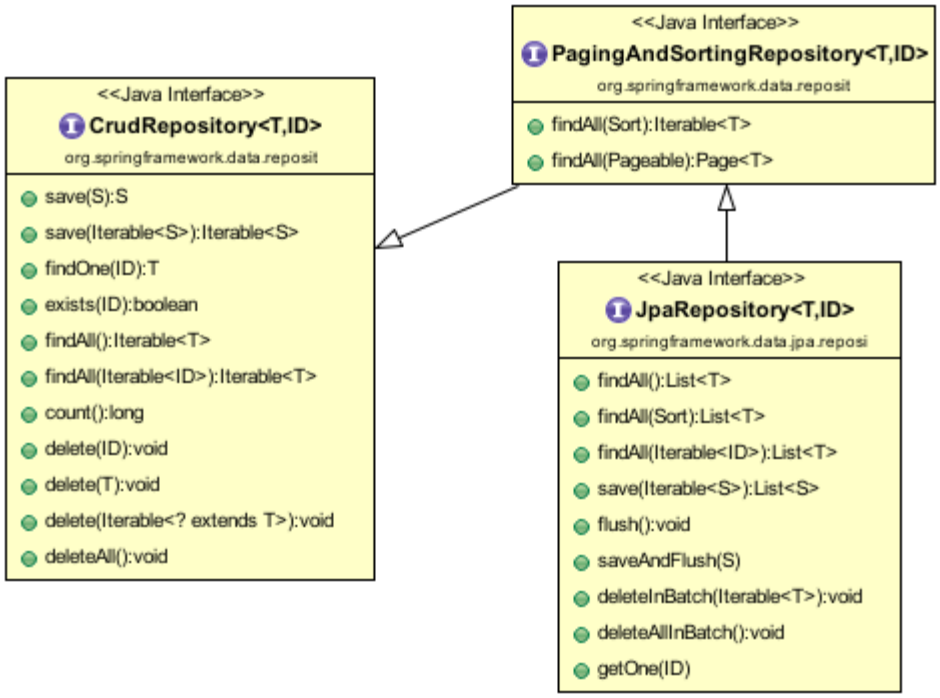
```
<bean class=
"org.springframework.dao.annotation.PersistenceExceptionTranslationPostProcessor"/>
```

Este bean, se encarga de capturar todas las excepciones lanzadas por clases anotadas con @repository y realizar la conversión.

Spring Data Jpa

Framework que extiende las funcionalidades de Spring ORM, permitiendo definir **Repositorios** de forma mas sencilla, sin repetir código, dado que ofrece numerosos métodos ya implementados y la posibilidad de crear nuevos tanto de consulta como de actualización de forma sencilla.

El framework, se basa en la definición de interfaces que extiendan la siguiente jerarquia, concretando el tipo entidad y el tipo de la clave primaria.



De forma paralela a las interfaces Jpa, existen para Mongo y Redis.

Por lo que la creación de un repositorio con Spring Data Jpa, se basará en la implementación de la interface **JpaRepository**

```
public interface CustomerRepository extends JpaRepository<Customer, Long> {}
```

El convenio indica que el nombre de la interface, será <**entidad**>**Repository**.

La magia de Spring Data, es que no es necesario definir las implementaciones, estas se crean en tiempo de ejecución según se defina el Repositorio.

Para que esto suceda, se tendrá que añadir a la configuración con JavaConfig

```
@EnableJpaRepositories(basePackages="com.cursospring.persistencia")
```

o para configuraciones con XML

```
<jpa:repositories base-package="com.cursospring.persistencia" />
```

Además Spring Data Jpa, exige la definición de un bean de tipo **AbstractEntityManagerFactoryBean** que se llame **entityManagerFactory**.

```
<bean id="entityManagerFactory" class=
"org.springframework.orm.jpa.LocalContainerEntityManagerFactoryBean">
    <property name="dataSource" ref="ds" />
    <property name="jpaVendorAdapter" ref="jpaVendorAdapter" />
    <property name="packagesToScan" value="com.curso.modelo.entidad"/>
    <property name="jpaProperties">
        <props>
            <prop key="hibernate.dialect">org.hibernate.dialect.DerbyDialect</prop>
            <prop key="hibernate.show_sql">true</prop>
            <prop key="hibernate.format_sql">true</prop>
            <prop key="hibernate.hbm2ddl.auto">validate</prop>
            <prop key="hibernate.default_schema">CLIENTES</prop>
        </props>
    </property>
</bean>
```

Y otro de tipo **TransactionManager** que se llame **transactionManager**

```
<bean id="transactionManager" class="org.springframework.orm.jpa.JpaTransactionManager">
    <property name="entityManagerFactory" ref="entityManagerFactory" />
</bean>
```

Ya que son dependencias de la implementación autogenerada que proporciona Spring Data.

Querys Personalizadas

Se pueden extender los métodos que ofrezca el Repositorio, siguiendo las siguientes reglas

- Prefijo del nombre del método **findBy** para busquedas y **countBy** para conteo de coincidencias.
- Seguido del nombre de los campos de busqueda concatenados por los operadores correspondientes: And, Or, Between, ... Todos los operadores aquí

```
List<Person> findByLastname(String lastname);

List<Person> findByEmailAddressAndLastname(EmailAddress emailAddress, String lastname);

List<Person> findDistinctPeopleByLastnameOrFirstname(String lastname, String firstname);
List<Person> findPeopleDistinctByLastnameOrFirstname(String lastname, String firstname);

List<Person> findByLastnameIgnoreCase(String lastname);

List<Person> findByLastnameAndFirstnameAllIgnoreCase(String lastname, String firstname);

List<Person> findByLastnameOrderByFirstnameAsc(String lastname);
List<Person> findByLastnameOrderByFirstnameDesc(String lastname);
```

Tambien se pueden definir consultas personalizadas con JPQL

```
@Query("from Country c where lower(c.name) like lower(?1)")
List<Country> getByNameWithQuery(String name);

@Query("from Country c where lower(c.name) like lower(?1)")
Country findByNameWithQuery(@Param("name") String name);

@Query("select case when (count(c) > 0)  then true else false end from Country c where
c.name = ?1)")
boolean exists(String name);
```

Paginación y Ordenación

De forma analoga a la definición

A las consultas se puede añadir un último parametro de tipo **Pageable** o **Sort**, que permite definir el criterio de paginación o de ordenación.

```
Country findByNameWithQuery(Integer population, Sort sort);

@Query("from Country c where lower(c.name) like lower(?1)")
Page<Country> findByNameWithQuery(String name, Pageable page);
```

Definiendose el criterio a aplicar en tiempo de ejecución

```
countryRepository.findByNameWithQuery("%i%", new Sort( new Sort.Order(Sort.Direction.ASC
,"name")));

Page<Country> page = countryRepository.findByNameWithQuery("%i%", new PageRequest(0, 3,
new Sort( new Sort.Order(Sort.Direction.ASC,"name"))));
```

Inserción / Actualización

Se pueden definir con JPQL, siempre que se cumpla

- El método debe estar anotado con @Modifying si no Spring Data JPA interpretará que se trata de una select y la ejecutará como tal.
- Se devolverá o void o un entero (int\Integer) que contendrá el número de objetos modificados o eliminados.
- El método deberá ser transaccional o bien ser invocado desde otro que sí lo sea.

```
@Transactional
@Modifying
@Query("UPDATE Country set creation = (?1)"
int updateCreation(Calendar creation);
```

```
@Transactional
int deleteByName(String name);
```

Se han de declarar en la clase **@Entity** con las anotaciones

- **@NamedStoredProcedureQueries**
- **@NamedStoredProcedureQuery**
- **@StoredProcedureParameter**

```
@Entity
@Table(name = "MYTABLE")
@NamedStoredProcedureQueries({
    @NamedStoredProcedureQuery(
        name = "in_only_test",
        procedureName = "test_pkg.in_only_test",
        parameters = {
            @StoredProcedureParameter(
                mode = ParameterMode.IN,
                name = "inParam1",
                type = String.class)
        }
    ), @NamedStoredProcedureQuery(
        name = "in_and_out_test",
        procedureName = "test_pkg.in_and_out_test",
        parameters = {
            @StoredProcedureParameter(
                mode = ParameterMode.IN,
                name = "inParam1",
                type = String.class),
            @StoredProcedureParameter(
                mode = ParameterMode.OUT,
                name = "outParam1",
                type = String.class)
        })
})
public class MyTable implements Serializable { }
```

Y para emplearlas en los repositorios de **JPA Data**, se han de utilizar las siguientes anotaciones

- **@Procedure**
- **@Param**

```
public interface MyTableRepository extends CrudRepository<MyTable, Long> {

    @Procedure(name = "in_only_test")
    void inOnlyTest(@Param("inParam1") String inParam1);

    @Procedure(name = "in_and_out_test")
    String inAndOutTest(@Param("inParam1") String inParam1);
}
```

Spring Boot

Si se emplea con Spring Boot, unicamente con añadir el starter **spring-boot-starter-data-jpa**

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-data-jpa</artifactId>
</dependency>
```

Se tendrán configurados los objetos de JPA necesarios en el contexto de Spring y se buscarán las interfaces que hereden de **Repository** en los subpaquetes del paquete base de la aplicacion Spring Boot.

Por defecto se trabaja con una base de datos H2 en local, con los siguientes datos

- DriverClass → org.h2.Driver
- JDBC URL → jdbc:h2:mem:testdb
- UserName → sa
- Password → <blank>

Se puede emplear una pequeña aplicación web que se incluye con el driver de H2 para acceder a esta base de datos y realizar pequeñas tareas de administracion, para ello hay que añadir la dependencia de Maven

```
<dependency>
    <groupId>com.h2database</groupId>
    <artifactId>h2</artifactId>
    <!-- <scope>runtime</scope> -->
</dependency>
```

NOTE

Al ser una consola Web, se ha de añadir tambien el starter web

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-web</artifactId>
</dependency>
```

Una vez añadido, se ha de regsitrar el Servlet **org.h2.server.web.WebServlet** que se proporciona en dicho jar, que es el que da acceso a la aplicacion de gestion de H2.

Una alternativa para registrar el servlet es a través del **ServletRegistrationBean** *de Spring*

```
@Configuration
public class H2Configuration {
    @Bean
    ServletRegistrationBean h2servletRegistration() {
        ServletRegistrationBean registrationBean = new ServletRegistrationBean(new
WebServlet(), "/console/*");
        return registrationBean;
    }
}
```

Query DSL

Para poder emplear este API, se han de añadir las siguientes dependencias Maven

```
<dependency>
    <groupId>com.querydsl</groupId>
    <artifactId>querydsl-apt</artifactId>
    <version>4.1.4</version>
    <scope>provided</scope>
</dependency>

<dependency>
    <groupId>com.querydsl</groupId>
    <artifactId>querydsl-jpa</artifactId>
    <version>4.1.4</version>
</dependency>
```

En Spring Boot, ya se contempla esta libreria, por lo que unicamente habrá que añadir

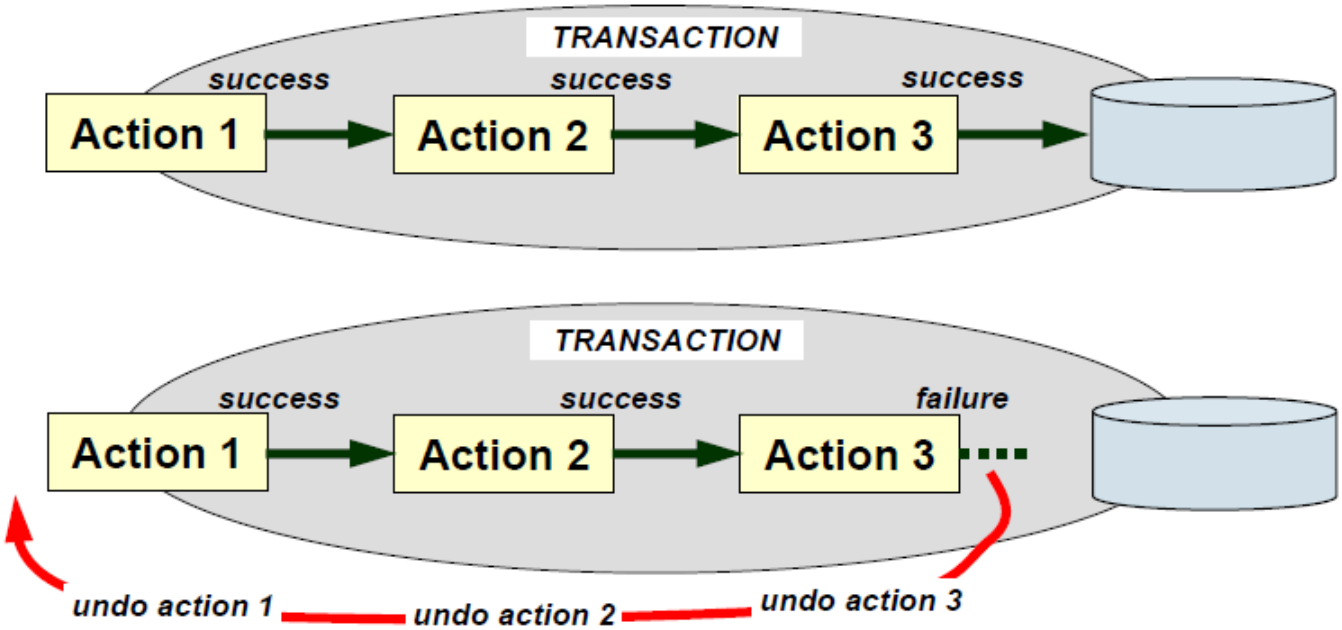
```
<dependency>
    <groupId>com.querydsl</groupId>
    <artifactId>querydsl-apt</artifactId>
</dependency>

<dependency>
    <groupId>com.querydsl</groupId>
    <artifactId>querydsl-jpa</artifactId>
</dependency>
```

Transacciones

Una transacción, es una operación de todo o nada, compuesta por una serie de suboperaciones, que se han de llevar a cabo como una unidad o no realizarse ninguna.

Características de las Transacciones (ACID)

- **Atómicas**: Todas las suboperaciones de la transacción se realizan como una unidad, si alguna no se puede llevar a cabo, no se realiza ninguna.
- **Coherentes**: Una vez finalizada la transacción, ya sea con éxito o no, el sistema queda en un estado coherente.
- **Independientes**: Deben de estar aisladas unas de otras, evitando lecturas y escrituras simultaneas.
- **Duraderas**: Los resultados de la transacción, deben ser persistidos, evitando su perdida por un fallo del sistema.



Spring es compatible con la gestión de transacciones de forma declarativa y programática.

La gestión programática, permite mayor precisión a la hora de establecer los limites de la transacción, ya que la unidad de operación de la transacción en este caso es la sentencia

La gestión declarativa, es menos precisa ya que la unidad de operación será el método, pero es más limpia ya que permite desacoplar un requisito de su comportamiento transaccional.

Para transacciones no distribuidas, cuyo ambito es un único recurso transaccional, por ejemplo la base de datos de clientes, Spring permite emplear como gestor de la transacción, el que incluye el proveedor de persistencia, Hibernate, OpenJpa, EclipseLink, …., sea este gestor JTA, o no.

Para transacciones distribuidas, se empleara una implementación de JTA.

La forma de adaptar la implementación especifica al contexto de Spring, será a través de objetos **PlatformTransactionManager** proporcionados por Spring

Spring proporciona entre otras las siguientes implementaciones

- CciLocalTransactionManager
- DataSourceTransactionManager
- HibernateTransactionManager
- JdoTransactionManager
- JpaTransactionManager
- JtaTransactionManager
- JmsTransactionManager
- WeblogicJtaTransactionManager, ...

Transacciones con JDBC

En el caso de JDBC puro, se empleará **DataSourceTransactionManager**, del cual se definirá un Bean

En XML

```
<bean id="transactionManager" class=
"org.springframework.jdbc.datasource.DataSourceTransactionManager">
    <property name="dataSource" ref="ds"/>
</bean>
```

En JavaConfig

```
@Bean
public DataSourceTransactionManager transactionManager(DataSource dataSource){
    return new DataSourceTransactionManager(dataSource);
}
```

Transacciones con Hibernate

En el caso de Hibernate, se empleará **HibernateTransactionManager**, del cual se definirá un Bean

En el XML

```
<bean id="transactionManager" class=
"org.springframework.orm.hibernate5.HibernateTransactionManager">
    <property name="sessionFactory" ref="sessionFactory" />
</bean>
```

En JavaConfig

```
@Bean
public HibernateTransactionManager transactionManager(SessionFactory sessionFactory){
    return new HibernateTransactionManager(sessionFactory);
}
```

Transacciones con Jpa

En el caso de Jpa, se empleará **JpaTransactionManager**, del cual se definirá un Bean

En el XML

```
<bean id="transactionManager" class="org.springframework.orm.jpa.JpaTransactionManager">
    <property name="entityManagerFactory" ref="entityManagerFactory" />
</bean>
```

En JavaConfig

```
@Bean
public JpaTransactionManager transactionManager(EntityManagerFactory
entityManagerFactory){
    return new JpaTransactionManager(entityManagerFactory);
}
```

Transacciones con Jta

En el caso de Jta, se empleará **JtaTransactionManager**, del cual se definirá un Bean

En el XML

```
<bean id="transactionManager" class=
"org.springframework.transaction.jta.JtaTransactionManager">
    <propertyname="transactionManagerName"  value="java:/TransactionManager"/>
</bean>
```

En JavaConfig

```
@Bean
public JtaTransactionManager transactionManager(){
    JtaTransactionManager transactionManager = new JtaTransactionManager();
    transactionManager.setUserTransactionName("java:comp/UserTransaction");
    return transactionManager;
}
```

Transacciones programaticas

Spring proporciona **TransactionTemplate**, una clase que permite manejar transacciones programáticas.

Esta clase necesitará cualquiera de las implementaciones de **TransactionManager**.

```
<bean id="transactionTemplate" class=
"org.springframework.transaction.support.TransactionTemplate">
    <property name="transactionManager" ref="transactionManager" />
</bean>
```

Para emplearla, se deberá incluir como dependencia en aquellos Bean que precisen manejar el comportamiento transaccional, los **commit** y **rollback**

```
<bean id="MiServicio" class="com.servicio.MiServicioImpl">
    <property name="transactionTemplate" ref="transactionTemplate"/>
</bean>
```

La clase **TransactionTemplate** ofrece el método **execute(TransactionCallback)**, que permite definir el ambito en el que se realiza la transacción, todas las operaciones que se ejecuten dentro del **TransactionCallback** estarán dentro de la misma transacción.

Para trabajar con **TransactionCallback**, existen dos posibilidades dependiendo de si la transacción retorna un resultado o no, si retorna un resultado se ha de implementar la interace **TransactionCallback<T>** y si no retorna resultado se ha implementar la clase abstracta **TransactionCallbackWithoutResult**

```
txTemplate.execute(new TransactionCallbackWithoutResult() {
    @Override
    //Dentro de este método se ejecutan todas las operaciones que conformen la Tx
    protected void doInTransactionWithoutResult(TransactionStatus status) {
        em.persist(cliente);
    }
});
```

Si no se produce ningun error en la ejecucion de las sentencias que forman la transacción, se hará **commit** y si se produce algun error, se hará **rollback**.

Transacciones declarativas

Para configurar las transacciones se emplearán las etiquetas **@Transactional** aplicadas a métodos o clases.

```
@Transactional
public void altaCliente(Cliente cliente) {
    clienteDAO.insertar(cliente);
}
```

NOTE	Ojo que hay dos anotaciones una del estandar y otra de Spring, la de Spring es mas configurable, la del estandar precisa de otra anotacion **@TransactionAttribute** para configurar el comportamiento transaccional

Las cuales habrá que activar, para ello en XML

```
<tx:annotation-driven proxy-target-class="true" transaction-manager="transactionManager"
/>
```

Y en JavaConfig

```
@Configuration
@EnableTransactionManagement
public class Configuracion{}
```

Configuracion de Transacciones

La configuracion de las transacciones consiste en definir las **políticas transaccionales** de la transacción.

- **Propagation Behavior**. Limites de la transacción.
- **Isolation Level**. Nivel de aislamiento. Como la afectan otras transacciones.
- **Read Only**. En el caso de que sean de solo lectura para optimizar.
- **Timeout**. Máximo tiempo que es aceptable para la tx.
- **Normas de reversión**. Listado de excepciones que causan rollback y cuales no.

Propagation Behavior

Define los limites de la transacción, cuando empieza, cuando se pausa, cuando es obligatoria, … puede ser:

- **PROPAGATION_MANDATORY**: La ejecución del método debe realizarse en una transacción. Si no la hay, se lanza una excepción.
- **PROPAGATION_NESTED**: La ejecución del método debe realizarse en una transacción anidada, no existe compatibilidad con todos los proveedores.
- **PROPAGATION_NEVER**: La ejecución del método NO debe realizarse en una transacción. Si hay una en curso se lanza una excepción.
- **PROPAGATION_NOT_SUPPORTED**: La ejecución del método NO debe realizarse en una transacción. Si hay una en curso esta se suspende.
- **PROPAGATION_REQUIRED**: La ejecución del método debe realizarse en una transacción. Si no existe una en curso, se genera una nueva.
- **PROPAGATION_REQUIRES_NEW**: La ejecución del método debe realizarse en una transacción propia, siempre se crea una transacción. Si hay una en curso se suspende.
- **PROPAGATION_SUPPORTS**: La ejecución del método no tiene porque realizarse en una transacción, pero si hay una en curso se ejecuta dentro de ella.

El valor por defecto es **PROPAGATION_REQUIRED**.

Isolation Level

Cuando varias transacciones, comparten datos, se pueden producir una serie de problemas.

- **Lectura de datos sucios**: Ocurre cuando se le permite a una transacción la lectura de una fila que ha sido modificada por otra transacción concurrente pero todavía no ha sido cometida.

Los datos leído por Tx1, han sido escritos por Tx2, pero todavía no son persistentes (commit).

Transacción 1

```sql
/* Query 1 */
SELECT edad FROM usuarios WHERE id = 1;
/* leerá 20 */
```

Transacción 2

```sql
/* Consulta 2 */
UPDATE usuarios SET edad = 21 WHERE id = 1;
/* No se hace commit */
```

```sql
/* Query 1 */
SELECT edad FROM usuarios WHERE id = 1;
/* leerá 21 */
```

```sql
ROLLBACK; /* LECTURA SUCIA basada en bloqueo */
```

- **Lectura no repetible**: Ocurre cuando en el curso de una transacción una fila se lee dos veces y los valores no coinciden.

Los datos obtenido por Tx1 al realizar la misma consulta no son iguales, dado que Tx2, los esta variando.

Transacción 1

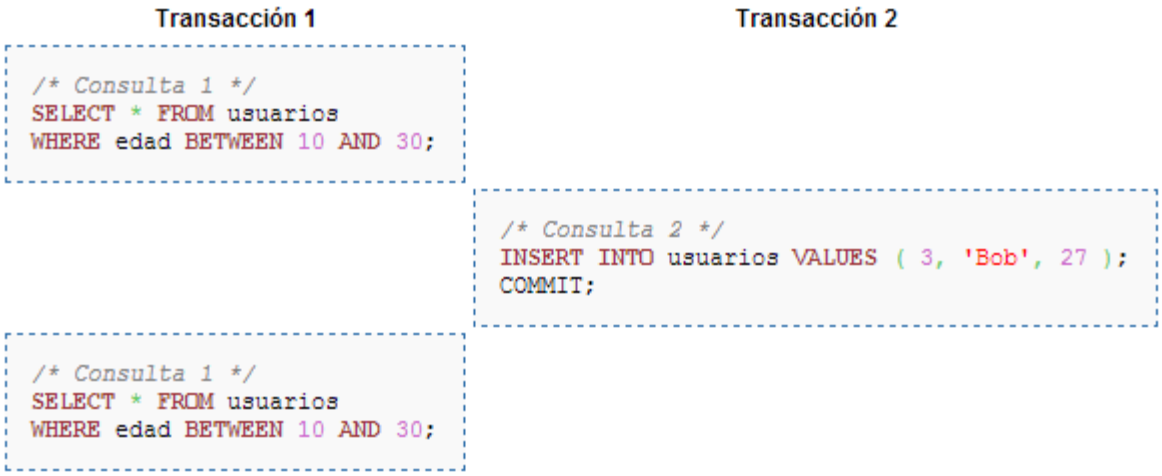
```sql
/* Consulta 1 */
SELECT * FROM usuarios WHERE id = 1;
```

Transacción 2

```sql
/* Consulta 2 */
UPDATE usuarios SET edad = 21 WHERE id = 1;
COMMIT; /* en MVCC o READ COMMITTED basado en bloqueos */
```

```sql
/* Consulta 1 */
SELECT * FROM usuarios WHERE id = 1;
COMMIT; /* REPEATABLE READ basado en bloqueos */
```

- **Lectura fantasma**: Ocurre cuando, durante una transacción, se ejecutan dos consultas idénticas, y los resultados de la segunda son distintos de los de la primera. Es una caso particular de las lecturas no repetibles



Para que esto no suceda, se puede recurrir a aislar los datos cuando estén siendo empleados por una transacción, bloqueándolos, pero esto produce una caída del rendimiento, por lo que hay que flexibilizar, permitiendo definir niveles de aislamiento.

Se definen los siguientes niveles de aislamiento

- **ISOLATION_DEFAULT**: Aplica el determinado por el almacén de datos al que accede.
- **ISOLATION_READ_UNCOMMITED**: Permite leer datos no consolidados.
- **ISOLATION_READ_COMMITED**: Permite leer datos de transacciones consolidadas. Evita lectura sucia.
- **ISOLATION_REPEATABLE_READ**: Varias lecturas del mismo campo, generan los mismos resultados excepto que la transacción lo modifique. Evita la lectura de datos sucios y la no repetible.
- **ISOLATION_SERIALIZABLE**: Aislamiento perfecto. Evita la lectura de datos sucios, la lectura no repetible y las lecturas fantasmas.

No todos los orígenes son compatibles con estos niveles.

El nivel por defecto es **ISOLATION_READ_COMMITED**.

Nivel de aislamiento	Lectura sucia	Lectura no repetibles	Lectura fantasma
Read Uncommitted	puede ocurrir	puede ocurrir	puede ocurrir
Read Committed	-	puede ocurrir	puede ocurrir
Repeatable Read	-	-	puede ocurrir
Serializable	-	-	-

Read Only

Si solo se van a realizar operaciones de solo lectura, Spring permite marcar esta opción, para llevar a cabo ciertas optimizaciones sobre la transacción.

Dado que esta optimización se aplica sobre una nueva transacción, tendrá sentido aplicarlo solo donde se inicien, esto es en transacciones cuya propagación sea:

- PROPAGATION_REQUIRED.
- PROPAGAION_REQUIRES_NEW.
- PROPAGATION_NESTED.

Si se trabaja con hibernate, esto implica que se empelara el modo de vaciado FLUSH_NEVER, que permite que Hibernate no sincronice con la base de datos hasta el final.

Timeout

Cuando se quiere limitar el tiempo durante el cual, la transacción puede tener bloqueados recursos del entorno de persistencia, será necesario especificar un tiempo máximo de espera, esto solo tendrá sentido donde se inicien, esto es en transacciones cuya propagación sea:

- PROPAGATION_REQUIRED.
- PROPAGATION_REQUIRES_NEW.
- PROPAGATION_NESTED.

Rollback

Conjunto de normas que definen cuando una excepción causa una reversión.

De forma predeterminada, solo se revierten las transacciones cuando hay excepciones en tiempo de ejecución (RuntimeException) y no con las comprobadas, al igual que con los EJB., aunque este comportamiento es configurable.

Spring Web

Framework que permite incluir el contexto de Spring en una aplicación web.

Se basa en la definición de un **Listener** de contexto web, que cree el contexto de Spring.

Para configuraciones tradicionales con **web.xml**, se define el **Listener** y la ubicación de todos los ficheros que formen el contexto de Spring.

```
<listener>
    <listener-class>
        org.springframework.web.context.ContextLoaderListener
    </listener-class>
</listener>
<context-param>
    <param-name>contextConfigLocation</param-name>
    <param-value>
        classpath:xfiles-config.xml,
        /WEB-INF/security.xml
    </param-value>
</context-param>
```

Para configuraciones basadas en JavaConfig, el Api proporciona una interface **WebApplicationInitializer**, que permtite la sustitución del **web.xml**

```
public class AppInitializer implements WebApplicationInitializer {
    @Override
    public void onStartup(ServletContext servletContext) throws ServletException {
        WebApplicationContext context = getContext();
        servletContext.addListener(new ContextLoaderListener(context));
    }

    private AnnotationConfigWebApplicationContext getContext() {
        AnnotationConfigWebApplicationContext context = new
AnnotationConfigWebApplicationContext();
        context.setConfigLocation("com.cursospring");
        return context;
    }
}
```

En este caso el contexto de Spring tambien se basa en JavaConfig.

Una vez definido el contexto de Spring y creado en la creación del contexto Web, cualquier componente Web JEE, podrá acceder a dicho contexto a través del contexto Web, de la siguiente manera

```
ApplicationContext context = WebApplicationContextUtils.getWebApplicationContext
(getServletContext());
```

Ambitos

A parte de los ambitos conocidos **singleton** y **prototype**, se introducen otros 2 scope

- **request** → Los Bean se crearán por cada nueva petición a la aplicación que lo precise.
- **session** → Los Bean se crearán por cada nueva sesión de usuario que lo precise.

Para asociarlos a un Bean se puede emplear la anotación **@Scope**

Recursos JNDI del Servidor

Es habitual que el Servidor de Aplicaciones donde se despliega la aplicación web, provea a esta de recursos empleando el api JNDI, desde Spring se puede incluir dichos recursos en el contexto de spring con la siguiente configuracion xml

```
<beans:bean id="dataSource" class="org.springframework.jndi.JndiObjectFactoryBean">
    <beans:property name="jndiName" value="java:comp/env/jdbc/MyLocalDB"/>
</beans:bean>
```

O de forma mas sencilla empleando el espacio de nombres jee

```
<jee:jndi-lookup expected-type="javax.sql.DataSource" id="dataSource" jndi-name=
"jdbc/MyLocalDB"/>
```

o JavaConfig

```
@Bean
public DataSource dataSource(@Value("${db.jndi}" String jndiName) {
    JndiDataSourceLookup lookup = new JndiDataSourceLookup();
    lookup.setResourceRef(true);
    return lookup.getDataSource(jndiName);
}
```

NOTE	La clase especializada en DataSource, **JndiDataSourceLookup**, se obtiene con **spring-jdbc**
NOTE	Donde db.jndi, valdra algo como **java:comp/env/jdbc/MyDB**

Para publicar el recurso JNDI se ha de consultar la documentación del servidor en concreto, dado que cada uno lo hace de una forma distinta.

Por ejemplo para un Tomcat 8, se haría definiciendo en **<TOMCAT_HOME>/conf/context.xml**

```
<Context>
  <Resource name="jdbc/MyDB" auth="Container" type="javax.sql.DataSource"
              maxTotal="100" maxIdle="30" maxWaitMillis="10000"
              username="admin" password="admin" driverClassName=
"org.apache.derby.jdbc.ClientDriver"
              url="jdbc:derby://localhost:1527/jndi"/>
</Context>
```

Mapeando dicho recurso en el fichero **/WEB-INF/web.xml**

```
<resource-ref>
  <description>DB Connection</description>
  <res-ref-name>jdbc/MyDB</res-ref-name>
  <res-type>javax.sql.DataSource</res-type>
  <res-auth>Container</res-auth>
</resource-ref>
```

En configuraciones JavaConfig, se puede añadir al **@Bean** la anotacion **@Resource** del api de servlets

```
@Configuration
public class Configuracion {
  @Bean
  @Resource(name="jdbc/MyDB")
  public DataSource dataSourceLookup() {
      final JndiDataSourceLookup dsLookup = new JndiDataSourceLookup();
      dsLookup.setResourceRef(true);
      DataSource dataSource = dsLookup.getDataSource("java:comp/env/jdbc/MyDB");
      return dataSource;
  }
}
```

Filtros

Como se ha comentado Spring Web, pretende extender el contenedor web estandar con los Bean del contexto de Spring, se ha visto como a traves del **ContextLoaderListener**, se crea el contexto y se permite el acceso a dicho Contexto desde componente dentro del contenedor web.

Cuando se trabaja con Spring MVC, se realiza algo parecido, ya que se asocia un contexto de Spring a un Servlet.

Tambien es posible extender los Filtros Web, Spring proporciona una implementación de Filtro Web **DelegatingFilterProxy** que permite delegar las intercepciones que realice dicho Filtro Web sobre Beans de Spring que implementen la interface **Filter**, en concreto delega la petición, sobre un Bean de

Spring, que se llame como el Filtro Web **DelegatingFilterProxy**.

Con XML

```
<filter>
  <filter-name>elNombreDeMiBeanEnSpring</filter-name>
  <filter-class>org.springframework.web.filter.DelegatingFilterProxy</filter-class>
</filter>
<filter-mapping>
  <filter-name>elNombreDeMiBeanEnSpring</filter-name>
  <url-pattern>/*</url-pattern>
</filter-mapping>
```

Con Java Config

```
public class AppInitializer implements WebApplicationInitializer {

    @Override
    public void onStartup(ServletContext servletContext) throws ServletException {
    // ...
    servletContext.addFilter("elNombreDeMiBeanEnSpring",
      new DelegatingFilterProxy("elNombreDeMiBeanEnSpring"))
        .addMappingForUrlPatterns(null, false, "/*");
    // ...
  }
}
```

En los ejemplos anteriores, deberá existir en el contexto de Spring un Bean llamado **elNombreDeMiBeanEnSpring**, que implemente la interface **Filter** del API de Servlets.

Spring MVC

Introduccion

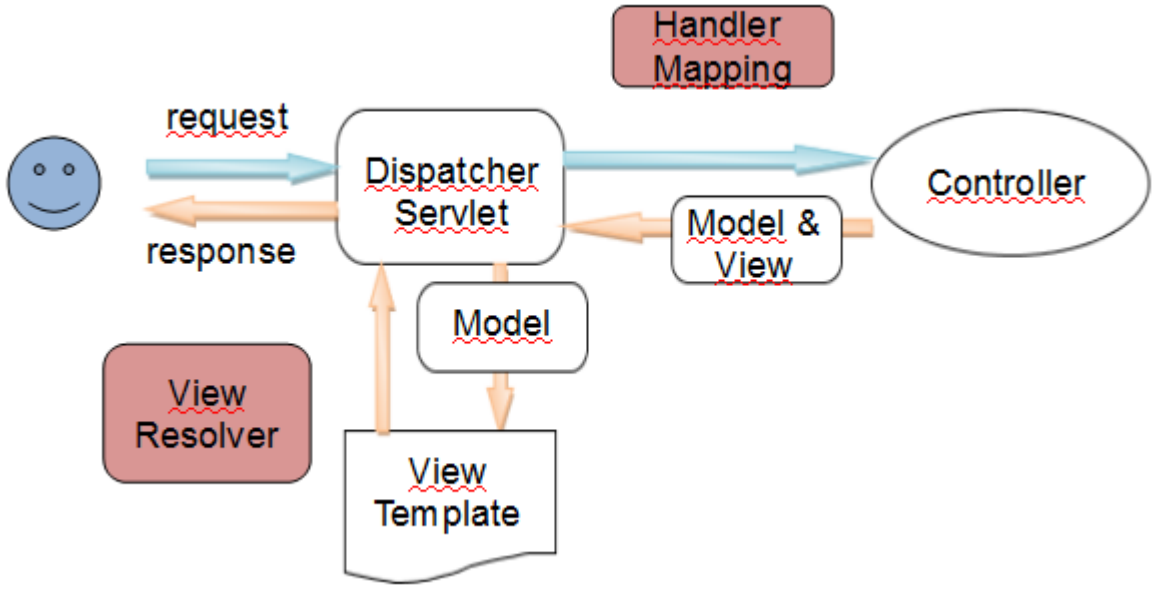
Spring MVC, como su nombre indica es un framework que implementa Modelo-Vista-Controlador, esto quiere decir que proporcionará componentes especializados en cada una de esas tareas.

Para incorporar las librerias con Maven, se añade al pom.xml

```
<dependency>
    <groupId>org.springframework</groupId>
    <artifactId>spring-webmvc</artifactId>
    <version>4.2.3.RELEASE</version>
</dependency>
```

Arquitectura

Spring MVC, como la mayoría de frameworks MVC, se basa en el patrón **FrontController**, en este caso el componente que realiza esta tarea es **DispatcherServlet**.



DispatcherServlet

El **DispatcherServlet**, realiza las siguientes tareas.

- Consulta con los **HandlerMapping**, que **Controller** ha de resolver la petición.
- Una vez el **HandlerMapping** le retorna que **Controller** ha de invocar, lo invoca para que resuelva la petición.
- Recoge los datos del **Model** que le envía el **Controller** como respuesta y el identificador de la **View** (o la propia **View** dependerá de la implementación del **Controller**) que se empleará para mostrar dichos datos.
- Consulta a la cadena de **ViewResolver** cual es la **View** a emplear, basandose en el identificador que le ha retornado el **Controller**.
- Procesa la **View** y el resultado lo retorna como resultado de la petición.

La configuración del **DispatcherServlet** se puede realizar siguiendo dos formatos

- Con ficheros XML. Para ello se han de declarar el servlet en el **web.xml**

```
<servlet>
    <servlet-name>miApp</servlet-name>
    <servlet-class>org.springframework.web.servlet.DispatcherServlet</servlet-class>
    <init-param>
        <param-name>contextConfigLocation</param-name>
        <param-value>/WEB-INF/miApp-servlet.xml</param-value>
    </init-param>
    <load-on-startup>1</load-on-startup>
</servlet>
<servlet-mapping>
    <servlet-name>miApp</servlet-name>
    <url-pattern>/expedientesx/*</url-pattern>
</servlet-mapping>
```

NOTE

De no incluir el parametro de configuracion **contextConfigLocation** para el servlet, sera importante el nombre del servlet, ya que por defecto este buscara en el directorio WEB-INF, el xml de Spring con el nombre **<servlet-name>-servlet.xml** en este caso **miApp-servlet.xml**

Se puede incluir más de un fichero de configuracion de contexto, separandolos con comas.

- Con clases anotadas al estilo **JavaConfig**. Para ello el API proporciona una interface que se ha de implementar **WebApplicationInitializer** y allí se ha de registrar el servlet.

```
public class AppInitializer implements WebApplicationInitializer {

    @Override
    public void onStartup(ServletContext servletContext) throws ServletException {
        WebApplicationContext context = getContext();
        ServletRegistration.Dynamic dispatcher = servletContext.addServlet(
"DispatcherServlet", new DispatcherServlet(context));
        dispatcher.setLoadOnStartup(1);
        dispatcher.addMapping("/");
    }

    private AnnotationConfigWebApplicationContext getContext() {
        AnnotationConfigWebApplicationContext context = new
AnnotationConfigWebApplicationContext();
        context.setConfigLocation(this.getClass().getPackage().getName());
        return context;
    }
}
```

ContextLoaderListener

Adicionalmente, se puede definir otro contexto de Spring global a la aplicación, para ello se ha de declarar el listener **ContextLoaderListener**, que al igual que el **DispatcherServlet** puede ser declarado de dos formas.

- Con XML

```
<context-param>
    <param-name>contextConfigLocation</param-name>
    <param-value>
        classpath:aplicacion.xml,
        /WEB-INF/seguridad.xml
    </param-value>
</context-param>
<listener>
    <listener-class>org.springframework.web.context.ContextLoaderListener</listener-
class>
</listener>
```

NOTE	Se puede incluir más de un fichero de configuracion de contexto, separandolos con comas.

- Con JavaConfig

```
public class AppInitializer implements WebApplicationInitializer {

    @Override
    public void onStartup(ServletContext servletContext) throws ServletException {

        WebApplicationContext context = getContext();
        servletContext.addListener(new ContextLoaderListener(context));
    }

    private AnnotationConfigWebApplicationContext getContext() {
        AnnotationConfigWebApplicationContext context = new
AnnotationConfigWebApplicationContext();
        context.setConfigLocation("expedientesx.cfg");
        return context;
    }
}
```

NOTE	La clase **AnnotationConfigWebApplicationContext** es una clase capaz de descubrir y considerar los Beans declarados en clases anotadas con **@Configuration**

Namespace MVC

Se incluye el siguiente namespace con algunas etiquetas nuevas, que favorecen la configuracion del contexto

```
xmlns:mvc="http://www.springframework.org/schema/mvc"
```

ResourceHandler (Acceso a recursos directamente)

No todas las peticiones que se realizan a la aplicación necesitarán que se ejecute un **Controller**, algunas de ellas harán referencia a imagenes, hojas de estilo, ... Se puede añadir con XML o JavaConfig

Con XML

```
<mvc:resources mapping="/resources/**" location="/resources/" />
```

Donde **mapping** hace referencia al patrón de URL de la petición y **location** al directorio dentro de **src/main/webapp** donde encontrar los recursos.

NOTE	La forma de abordar esta explicación, es retomar la arquitectura y el patrón **FrontController**, y la no necesidad de un **Controller** para ofrecer un recurso estatico, los **Controller** son necesarios para los recursos dinamicos, para los estaticos introducen demasida complejidad de forma inecesaria.

Con JavaConfig, se ha de hacer extender la clase **@Configuration** de **WebMvcConfigurerAdapter** y sobrescribir el método **addResourceHandlers** con lo siguiente.

```
@Override
public void addResourceHandlers(final ResourceHandlerRegistry registry) {
    registry.addResourceHandler("/resources/**").addResourceLocations("/resources/");
}
```

Donde **ResourceHandler** hace referencia al patrón de URL de la petición y **ResourceLocation** al directorio donde encontrar los recursos.

NOTE	La forma de abordar esta explicación, es retomar la arquitectura y el patrón **FrontController**, y la no necesidad de un **Controller** para ofrecer un recurso estatico, los **Controller** son necesarios para los recursos dinamicos, para los estaticos introducen demasida complejidad de forma inecesaria.

Default Servlet Handler

Cuando los recursos estaticos, estan situados en la carpeta **webapp**, se pueden sustituir las configuraciones anteriores por

```
<mvc:default-servlet-handler/>
```

o

```
@Configuration
@EnableWebMvc
public class WebConfig extends WebMvcConfigurerAdapter {
    @Override
    public void configureDefaultServletHandling(DefaultServletHandlerConfigurer
configurer) {
        configurer.enable();
    }
}
```

ViewController (Asignar URL a View)

En ocasiones se necesita acceder a una **View** directamente, sin pasar por un controlador, para ello Srping MVC ofrece los **ViewControllers**. Se puede añadir con XML o JavaConfig

Con XML

```
<mvc:view-controller path="/" view-name="welcome" />
```

Con JavaConfig, de nuevo se ha de hacer extender la clase **@Configuration** de la clase **WebMvcConfigurerAdapter**, en este caso implementando el método

```
@Override
public void addViewControllers(ViewControllerRegistry registry) {
    registry.addViewController("/").setViewName("index");
}
```

En este caso **ViewController** representa el path que le llega al **DispatcherServlet** y **ViewName** el nombre de la **View** que deberá ser resuelto por un **ViewResolver**.

NOTE	Los **ViewController** se resuelven posteriormente a los **Controller** anotados con **RequestMapping**, por lo que si se emplean mappings con path similares en ambos escenarios, nunca se llegará a los **ViewController**, para conseguirlo se ha de configurar la precedencia del **ViewControllerRegistry** a un valor inferior al del **RequestMappingHandlerMapping**.
NOTE	Los **ViewController** no pueden acceder a elementos del Modelo definidos con **@ModelAttribute**, ya que estos son interpretados por el **RequestMappingHandlerMapping**, que no participa en el proceso de resolución de los **ViewController**

HandlerMapping

Es el primero de los componentes necesarios dentro del flujo de Spring MVC, siendo el encargado de encontrar el controlador capaz de procesar la petición recibida.

Este componente extrae de la URL un Path, que coteja con las entradas configuradas dependiendo de la implementación empleada.

Para activar los HandlerMapping unicamente hay que declararlos en el contexto de Spring como Beans.

NOTE	Dado que se pueden configurar varios **HandlerMapping**, para establecer en que orden se han de emplear, existe la propiedad **Order**

El API proporciona las siguientes implementaciones

- **BeanNameUrlHandlerMapping** → Usa el nombre del Bean **Controller** como mapeo **<bean name="/inicio.htm" ...** >, debe comenzar por /.
- **SimpleUrlHandlerMapping** → Mapea mediante propiedades **<prop key="/verClientes.htm">beanControlador</prop>**
- **ControllerClassNameHandlerMapping** → Usa el nombre de la clase asumiendo que termina en **Controller** y sustituyéndo la palabra **Controller** por **.htm**
- **DefaultAnnotationHandlerMapping** → Emplea la propiedad path de la anotación **@RequestMapping**

NOTE	Las implementaciones por defecto en Spring MVC 3 son **BeanNameUrlHandlerMapping** y **DefaultAnnotationHandlerMapping**

BeanNameUrlHandlerMapping

Al emplear esta configuración, cuando lleguen peticiones con path **/helloWoorld.html**, el **Controller** que lo procesará será de tipo **EjemploAbstractController**

```
<bean class="org.springframework.web.servlet.handler.BeanNameUrlHandlerMapping" />

<bean name="/helloWorld.html" class="org.ejemplos.springmvc.HelloWorldController" />
```

SimpleUrlHandlerMapping

```
<bean class="org.springframework.web.servlet.handler.SimpleUrlHandlerMapping">
    <property name="mappings">
    <props>
        <prop key="/helloWorld.htm">helloWorldController</prop>
    </props>
    </property>
</bean>
<bean name="helloWorldController" class="org.ejemplos.springmvc.HelloWorldController" />
```

ControllerClassNameHandlerMapping

```
public class HelloWorldController extend AbstractController{
    @Override
    protected ModelAndView handleRequestInternal(HttpServletRequest request,
HttpServletResponse response)
            throws Exception {
        return new ModelAndView("otro");
    }
}
```

DefaultAnnotationHandlerMapping

```
@RequestMapping("helloWorld")
public class HelloWorldController extend AbstractController{
    @Override
    protected ModelAndView handleRequestInternal(HttpServletRequest request,
HttpServletResponse response)
            throws Exception {
        return new ModelAndView("otro");
    }
}
```

RequestMappingHandlerMapping

Esta implementacion permite interpretar las anotaciones **@RequestMapping** en los controladores, haciendo coincidir la url, con el atributo **path** de dichas anotaciones.

```
@RequestMapping("helloWorld")
public class HelloWorldController extend AbstractController{
    @Override
    protected ModelAndView handleRequestInternal(HttpServletRequest request,
HttpServletResponse response)
            throws Exception {
        return new ModelAndView("otro");
    }
}
```

El espacio de nombres **mvc**, ofrece una etiqueta que simplifica la configuracion

```
<mvc:annotation-driven/>
```

Tambien se ofrece una anotacion **@EnableWebMvc** a añadir a la clase **@Configuration** para la configuracion con JavaConfig, esta anotación, define por convencion una pila de **HandlerMapping**, ya

que en realidad lo que hace es cargar la clase **WebMvcConfigurationSupport** como calse **@Configuration**, en esta clase se describen los **HandlerMapping** cargados.

```
@Configuration
@EnableWebMvc
public class ContextoGlobal {
}
```

NOTE	En la ultima version de Spring no es necesario añadirlo, la unica diferencia al añadirlo, es que se consideran menos path validos para cada **@RequestMapping** definido, con ella solo **/helloWorld** y sin ella **/helloWorld**, **/helloWorld.**\ * y **/helloWorld/**

Controller

El siguiente de los componentes en el que delega el **DispatcherServlet**, será el encargado de ejecutar la logica de negocio.

Spring proporciona las siguientes implementaciones

- **AbstractController**
- **ParametrizableViewController**
- **AbstractCommandController**
- **SimpleFormController**
- **AbstractWizardFormController**
- **@Controller**

@Controller

Anotacion de Clase, que permite indicar que una clase contiene funcionalidades de Controlador.

```
@Controller
public class HelloWorldController {
    @RequestMapping("helloWorld")
    public String helloWorld(){
        return "exito";
    }
}
```

La firma de los métodos de la clase anotada es flexible, puede retornar

- String

- View
- ModelAndView
- Objeto (Anotado con @ResponseBody)

Si se desea que el retorno provoque una redireccion, basta con incluir el prefijo **redirect:**

```
@Controller
public class HelloWorldController {
    @RequestMapping("helloWorld")
    public String helloWorld(){
        return "redirect:/exito";
    }
}
```

NOTE	Cuando se retorna el id de una View, se hace participe a esa View de la actual Request, cuando se redirecciona, se crea una Request nueva.

Y puede recibir como parámetro

- **Model** → Datos a emplear en la **View**.
- **Map<String, Object> model** → Lo resuelve como Model, permite el desacoplamiento con el API de Spring.
- Parametros anotados con **@PathVariable** → Dato que llega en el path de la Url.
- Parametros anotados con **@RequestParam** → Dato que llega en los parametros de la Url.
- Parametros anotados con **@CookieValue** → Dato que llega en un HTTP cookie
- Parametros anotados con **@RequestHeader** → Dato que llega en un HTTP Header
- Parametros anotados com **@SessionAttribute** → Atributo de la Sesion Http que se desea inyectar en el controlador
- **HttpServletRequest**
- **HttpServletResponse**
- **HttpSession**
- **Locale**
- **Principal**
- **Errors**
- **BindingResult**
- **UriComponentsBuilder**

Activación de @Controller

Para activar esta anotación, habra que indicarle al contexto de Spring a partir de que paquete puede buscarla. Se puede hacer con XML y con JavaConfig

Con XML, se emplea la etiqueta **ComponentScan**

```
<context:component-scan base-package="controllers"/>
```

NOTE	Esta etiqueta activa el descubrimineto de las clases anotadas con @Component, @Repository, @Controller y @Service

Con JavaConfig, se emplea la anotacion **@ComponentScan**

```
@Configuration
@EnableWebMvc
@ComponentScan(basePackages={ "controllers" })
public class ContextoGlobal {

}
```

NOTE	Esta anotacion activa el descubrimineto de las clases anotadas con @Component, @Repository, @Controller y @Service

@RequestMapping

Es la anotacion que permite resolver si la **HttpRequest** llega o no a un controlador, define un filtro de seleccion de Controladores basado en las caracteristicas del Request.

Se pueden definir

- method → Method HTTP
- path → Url
- consumes → Corresponde a la cabecera Content-Type
- produces → Corresponde a la cabecera Accept
- params → Permite indicar la obligatoriedad de la presencia de un parametro (params="myParam"), asi como de su ausencia (params="!myParam") o un valor determinado (params="myParam=myValue")
- headers → Igual que la anterior pero para cabeceras

@PathVariable

Anotacion que permite obtener información de la url que provoca la ejecucion del controlador.

```
@RequestMapping(path="/saludar/{nombre}")
public ModelAndView saludar(@PathVariable("nombre") String nombre){
}
```

Para el anterior ejemplo, dada la siguiente url **http://...../saludar/Victor**, el valor del parametro **nombre**, será **Victor**

NOTE

Se pueden definir expresiones regulares para alimentar a los @PathVariable, siguiendo la firma **{varName:regex}**, por ejemplo

@RequestMapping("/spring-web/{symbolicName:[a-z-]}-{version:\\d\\.\\d\\.\\d}{extension:\\.[a-z]}") public void handle(@PathVariable String version, @PathVariable String extension) { // ... }

@RequestParam

Anotacion que permite obtener información de los parametros de la url que provoca la ejecucion del controlador.

```
@RequestMapping(path="/saludar")
public ModelAndView saludar(@RequestParam("nombre") String nombre){
}
```

Para el anterior ejemplo, dada la siguiente url **http://...../saludar?nombre=Victor**, el valor del parametro **nombre**, será **Victor**

@SessionAttribute

Anotacion que permite recibir en un método de controlador, un atributo insertado con anterioridad en la Sesion.

```
@GetMapping("/nuevaFactura")
public String nuevaFactura(@SessionAttribute("login") Login login, @ModelAttribute
Factura factura) {
    return "factura/formulario";
}
```

@RequestBody

Permite tranformar el contenido del **body** de peticiones **POST** o **PUT** a un objeto java, tipicamente una representación en JSON.

```
@RequestMapping(path="/alta", method=RequestMethod.POST)
public String getDescription(@RequestBody UserStats stats){
    return "resultado";
}

public class UserStats{
    private String firstName;
    private String lastName;
}
```

En el ejemplo anterior, se convertirán a objeto, contenidos del **body** de la petición como por ejemplo

```
{ "firstName" : "Elmer", "lastName" : "Fudd" }
```

Para transformaciones a JSON, se emplea la siguiente libreria de **Jackson**

```
<dependency>
    <groupId>org.codehaus.jackson</groupId>
    <artifactId>jackson-mapper-asl</artifactId>
    <version>1.4.2</version>
</dependency>
```

@ResponseBody

Analogo al anterior, pero para generar un resultado.

Se aplica sobre métodos que retornan un objeto de información.

```
// controller
@ResponseBody
@RequestMapping("/description")
public Description getDescription(@RequestBody UserStats stats){
    return new Description(stats.getFirstName() + " " + stats.getLastname() + " hates
wacky wabbits");
}

public class UserStats{
    private String firstName;
    private String lastName;
    // + getters, setters
}

public class Description{
    private String description;
    // + getters, setters, constructor
}
```

Precisa dar de alta el API de marshall en el classpath.

```
<dependency>
    <groupId>org.codehaus.jackson</groupId>
    <artifactId>jackson-mapper-asl</artifactId>
    <version>1.4.2</version>
</dependency>
```

Es muy empleado en servicios REST.

@ModelAttribute

Se pueden añadir Beans al objeto **Model** de un controlador en el ambito **request** con la anotación **@ModelAttribute**.

```
@Controller
public class MyController {

    @ModelAttribute("persona")
    public Persona addPersonaToModel() {
        return new Persona("Victor");
    }
}
```

@SessionAttributes

Tambien se puede asociar a la **session**, para ello se emplea la anotación **@SessionAttributes("nombreDelBeanDelModeloAAlmacenarEnLosAtributosDeLaSession")**, incluyendola como anotación de clase en la clase **Controller** que declare el bean del modelo con **@ModelAttribute**.

```
@Controller
@SessionAttributes("persona")
public class MyController {

    @ModelAttribute("persona")
    public Persona addPersonaToModel() {
        return new Persona("Victor");
    }
}
```

Los objetos en Model, pueden ser inyectados directamente en los métodos del controlador con **@ModelAttribute**

```
@RequestMapping("/saludar")
public String saludar (@ModelAttribute("persona") Persona persona, Model model) {
    return "exito";
}
```

@InitBinder

Permite redefinir:

- CustomFormatter → Permite defirnir transformaciones de tipos, se basa en la interface **Formatter**
- Validators → Validadores nuevos a aplicar a los Bean del Modelo, se basa en **Validator**
- CustomEditor → Parseos a aplicar a campos de los formularios, se basan en **PropertyEditor**

```
@InitBinder
public void customizeBinding(WebDataBinder binder) {

}
```

@ExceptionHandler

Permiten definir vistas a emplear cuando se producen excepciones en los métodos de control

```
@ExceptionHandler(CustomException.class)
public ModelAndView handleCustomException(CustomException ex) {

    ModelAndView model = new ModelAndView("error");
    model.addObject("ex", ex);
    return model;
}
```

@ControllerAdvice

Permiten definir en una clase independiente configuraciones de **@ExceptionHandler**, **@InitBinder** y **@ModelAttribute** que afectaran a los controladores que se desee, siempre que sean procesados por **RequestMappingHandlerMapping**, por ejemplo los **ViewControllers** no se ven afectados por esta funcionalidad.

```
@ControllerAdvice(basePackages="com.viewnext.holamundo.javaconfig.controllers")
public class GlobalConfig {
    @ModelAttribute
    public void initGlobal(Model model) {
        model.addAttribute("persona", new Persona());
    }
}
```

ViewResolver

El último componente a definir del flujo es el **ViewResolver**, este componente se encarga de resolver que **View** se ha emplear a partir del objeto **View** retornado por el **Controller**.

Pueden existir distintos **Bean** definidos de tipo **ViewResolver**, pudiendose ordenar con la propiedad **Order**.

NOTE	Es importante que de emplear el InternalResourceViewResolver, este sea el ultimo (Valor mas alto).

Se proporcionan varias implementaciones, alguna de ellas

- **InternalResourceViewResolver** → Es el más habitual, permite interpretar el **String** devuelto por el **Controller**, como parte de la url de un recurso, componiendo la URL con un prefijo y un sufijo. Aunque es configurable, emplea por defecto las **View** de tipo **InternalResourceView**, de emplearse **JstlView**, se necesitaria añadir al classpath la dependencia con **jstl**
- **BeanNameViewResolver** → Busca un **Bean** declarado de tipo **View** cuyo **Id** sea igual al **String** retornado por el **Controller**.

- **ContentNegotiatingViewResolver** → Delega en otros **ViewResolver** dependiendo del **ContentType**.
- **FreeMarkerViewResolver** → Similar al **InternalResourceViewResolver**, pero el recurso buscado debe ser una plantilla Freemarker.
- **JasperReportsViewResolver** → Similar al **InternalResourceViewResolver**, pero el recurso buscado debe ser una plantilla JasperReport.
- **ResourceBundleViewResolver** → Busca la implementacion de la View en un fichero de properties.
- **TilesViewResolver** → Busca una plantillas de **Tiles** con nombre igual al **String** retornado por el **Controller**
- **VelocityViewResolver** → Similar al **InternalResourceViewResolver**, pero el recurso buscado debe ser una plantilla Velocity.
- **XmlViewResolver** → Similar a **BeanNameViewResolver**, salvo porque los **Bean** de las **View** han de ser declaradas en el fichero **/WEB-INF/views.xml**
- **XsltViewResolver** → Similar al **InternalResourceViewResolver**, pero el recurso buscado debe ser una plantilla XSLT.

InternalResourceViewResolver

Se ha de definir el Bean

```
<bean class="org.springframework.web.servlet.view.InternalResourceViewResolver">
    <propertyname="prefix"value="/WEB-INF/views/"/>
    <propertyname="suffix"value=".jsp"/>
</bean>
```

XmlViewResolver

Se ha de definir el Bean

```
<bean class="org.springframework.web.servlet.view.XmlViewResolver">
    <property name="location" value="/WEB-INF/views.xml"/>
    <property name="order" value="0"/>
</bean>
```

Y en el fichero **/WEB-INF/views.xml**

```
<bean id="pdf/listado" class="com.aplicacion.presentacion.vistas.ListadoPdfView"/>
<bean id="excel/listado" class="com.aplicacion.presentacion.vistas.ListadoExcelView"/>
<bean id="json/listado" class=
"org.springframework.web.servlet.view.json.MappingJacksonJsonView"/>
```

ResourceBundleViewResolver

Se ha de definir el Bean

```
<bean id="viewResolver" class=
"org.springframework.web.servlet.view.ResourceBundleViewResolver">
    <property name="basename" value="views" />
</bean>
```

Y en el fichero **views.properties** que estará en la raiz del classpath.

```
listado.(class)=org.springframework.web.servlet.view.jasperreports.JasperReportsPdfView
listado.url=/WEB-INF/jasperTemplates/reporteAfines.jasper
listado.reportDataKey=listadoKey
```

Donde **url** y **reportDataKey**, son propiedades del objeto **JasperReportsPdfView**, y **listado** el **String** que retorna el **Controller**

View

Son los componentes que renderizaran la resuesta a la petición procesada por Spring MVC.

Existen diversas implementaciones dependiendo de la tecnologia encargada de renderizar.

- AbstractExcelView
- AbstractAtomFeedView
- AbstractRssFeedView
- MappingJackson2JsonView
- MappingJackson2XmlView
- AbstractPdfView
- AbstractJasperReportView
- AbstractPdfStamperView
- AbstractTemplateView
- InternalResourceView
- JstlView → Es la que se emplea habitualmente para los JSP, exige la libreria JSTL.
- TilesView
- XsltView

AbstractExcelView

El API de Spring proporciona una clase abstracta que esta destinada a hacer de puente entre el API capaz de generar un Excel y Spring, pero no genera el Excel, para ello hay que incluir una libreria como **POI**

```
<dependency>
    <groupId>org.apache.poi</groupId>
    <artifactId>poi</artifactId>
    <version>3.10.1</version>
</dependency>
```

Algunas de las clases que proporciona **POI** son

- HSSFWorkbook
- HSSFSheet
- HSSFRow
- HSSFCell

```
public class PoiExcellView extends AbstractExcelView {
    @Override
    protected void buildExcelDocument(Map<String, Object> model, HSSFWorkbook workbook,
HttpServletRequest request, HttpServletResponse response) throws Exception {
        // model es el objeto Model que viene del Controller
        List<Book> listBooks = (List<Book>) model.get("listBooks");
        // Crear una nueva hoja excel
        HSSFSheet sheet = workbook.createSheet("Java Books");
        sheet.setDefaultColumnWidth(30);
        HSSFRow header = sheet.createRow(0);
        header.createCell(0).setCellValue("Book Title");
        header.createCell(1).setCellValue("Author");
        int rowCount = 1;
        for (Book aBook : listBooks) {
            HSSFRow aRow = sheet.createRow(rowCount++);
            aRow.createCell(0).setCellValue(aBook.getTitle());
            aRow.createCell(1).setCellValue(aBook.getAuthor());
        }
        response.setHeader("Content-disposition", "attachment; filename=books.xls");
    }
}
```

AbstractPdfView

De forma analoga al anterior, para los PDF, se tiene la libreria **Lowagie**

```
<dependency>
    <groupId>com.lowagie</groupId>
    <artifactId>itext</artifactId>
    <version>4.2.1</version>
</dependency>
```

Algunas de las clases que proporciona **Lowagie** son

- Document
- PdfWriter
- Paragraph
- Table

```
public class ITextPdfView extends AbstractPdfView {
    @Override
    protected void buildPdfDocument(Map<String, Object> model, Document doc, PdfWriter
writer, HttpServletRequest request, HttpServletResponse response) throws Exception {
        // model es el objeto Model que viene del Controller
        List<Book> listBooks = (List<Book>) model.get("listBooks");
        doc.add(new Paragraph("Recommended books for Spring framework"));
        Table table = new Table(2);
        table.addCell("Book Title");
        table.addCell("Author");
            for (Book aBook : listBooks) {
            table.addCell(aBook.getTitle());
            table.addCell(aBook.getAuthor());
        }
        doc.add(table);
    }
}
```

JasperReportsPdfView

En este caso Spring proporciona una clase concreta, que es capaz de procesar las platillas de **JasperReports**, lo unico que necesita es la libreria de **JaspertReport**, la plantilla compilada **jasper** y un objeto **JRBeanCollectionDataSource** que contenga la información a representar en la plantilla.

NOTE	La plantilla sin compilar será un fichero **jrxml**, que es un xml editable.

```
<dependency>
    <groupId>jasperreports</groupId>
    <artifactId>jasperreports</artifactId>
    <version>3.5.3</version>
</dependency>
```

NOTE A tener en cuenta que la version de la libreria de JasperReport debe coincidir con la del programa iReport empleando para generar la plantilla.

```
<bean id="reporteAfines" class=
"org.springframework.web.servlet.view.jasperreports.JasperReportsPdfView">
    <property name="url" value="/WEB-INF/jasperTemplates/reporte.jasper"/>
    <property name="reportDataKey" value="listadoKey"></property>
</bean>
```

NOTE **reportDataKey** indica la clave dentro del objeto **Model** que referencia al objeto **JRBeanCollectionDataSource**

```
@Controller
public class AfinesReportController {
    @RequestMapping("/reporte")
    public String generarReporteAfines(Model model){
        JRBeanCollectionDataSource jrbean = new JRBeanCollectionDataSource(listado,
false);
        model.addAttribute("listadoKey", jrbean);
        return "reporteAfines";
    }
}
```

MappingJackson2JsonView

```
<dependency>
    <groupId>com.fasterxml.jackson.core</groupId>
    <artifactId>jackson-databind</artifactId>
    <version>2.4.1</version>
</dependency>
```

NOTE Es para versiones de Spring posteriores a 4, para la 3 se emplea otro API y la clase **MappingJacksonJsonView**

Los Bean a convertir a JSON, han de tener propiedades.

Formularios

Para trabajar con formularios Spring proporciona una libreria de etiquetas

```
<%@ taglibprefix="sf“ uri="http://www.springframework.org/tags/form"%>
```

Tag	Descripción
checkbox	Renders an HTML 'input' tag with type 'checkbox'.
checkboxes	Renders multiple HTML 'input' tags with type 'checkbox'.
errors	Renders field errors in an HTML 'span' tag.
form	Renders an HTML 'form' tag and exposes a binding path to inner tags for binding.
hidden	Renders an HTML 'input' tag with type 'hidden' using the bound value.
input	Renders an HTML 'input' tag with type 'text' using the bound value.
label	Renders a form field label in an HTML 'label' tag.
option	Renders a single HTML 'option'. Sets 'selected' as appropriate based on bound value.
options	Renders a list of HTML 'option' tags. Sets 'selected' as appropriate based on bound value.
password	Renders an HTML 'input' tag with type 'password' using the bound value.
radiobutton	Renders an HTML 'input' tag with type 'radio'.
select	Renders an HTML 'select' element. Supports databinding to the selected option.

Un ejemplo de definición de formulario podria ser

```
<form:form action="altaUsuario" modelAttribute="persona">
    <table>
        <tr>
            <td>Nombre:</td>
            <td><form:input path="nombre" /></td>
        </tr>
        <tr>
            <td>Apellidos:</td>
            <td><form:input path="apellidos" /></td>
        </tr>
        <tr>
            <td>Sexo:</td>
            <form:select path="sexo" items="${listadoSexos}"/>
        </tr>
        <tr>
            <td colspan="2">
            <input type="submit" value="Guardar info" />
            </td>
        </tr>
    </table>
</form:form>
```

NOTE	No es necesario definir el action si se emplea la misma url para cargar el formulario y para recibirlo, basta con cambiar unicamente el METHOD HTTP No hay diferencia entre **commandName** y **modelAtribute**

En el ejemplo anterior, se han definido a nivel del formulario.

- **action** → Indica la Url del Controlador.
- **modelAttribute** → Indica la clave con la que se envía el objeto que se representa en el formulario. (de forma analoga se puede emplear **commandName**)

Para recuperar en el controlador el objeto enviado, se emplea la anotación **@ModelAttribute**

El objeto que se representa en el formulario ha de existir al representar el formulario. Es típico para los formularios definir dos controladores uno GET y otro POST.

- El GET inicializara el objeto.
- El POST tratara el envío del formulario.

```
@RequestMapping(value="altaPersona", method=RequestMethod.GET)
public String inicializacionFormularioAltaPersonas(Model model){
    Persona p = new Persona(null, "", "", null, "Hombre", null);
    model.addAttribute("persona", p);
    model.addAttribute("listadoSexos", new String[]{"Hombre","Mujer"});
    return "formularioAltaPersona";
}
@RequestMapping(value="altaPersona", method=RequestMethod.POST)
public String procesarFormularioAltaPersonas(
    @ModelAttribute("persona") Persona p,  Model model){
        servicio.altaPersona(p);
        model.addAttribute("estado", "OK");
        model.addAttribute("persona", p);
        model.addAttribute("listadoSexos", new String[] {"Hombre","Mujer"});
        return "formularioAltaPersona";
}
```

Si se desea recibir un fichero desde el cliente, se empleará la tipologia **CommonsMultipartFile[]**

```
@RequestMapping(value = "/uploadFiles", method = RequestMethod.POST)
public String handleFileUpload(@RequestParam CommonsMultipartFile[] fileUpload) throws
Exception {
    for (CommonsMultipartFile aFile : fileUpload){
        // stores the uploaded file
        aFile.transferTo(new File(aFile.getOriginalFilename()));

    }
    return "Success";
}
```

Etiquetas

Spring proporciona dos librerias de etiquetas

- Formularios
- **<form:form></form:form>** → Crea una etiqueta HTML form.
- **<form:errors></form:errors>** → Permite la visualización de los errores asociados a los campos del **ModelAttribute**
- **<form:checkboxes items="" path=""/>** →

```
<form:checkbox path=""/>
```

```
<form:hidden path=""/>
```

```
<form:input path=""/>
```

```
<form:label path=""></form:label>
```

```
<form:textarea path=""/>
```

```
<form:password path=""/>
```

```
<form:radiobutton path=""/>
```

```
<form:radiobuttons path=""/>
```

```
<form:select path=""></form:select>
```

```
<form:option value=""></form:option>
```

```
<form:options/>
```

```
<form:button></form:button>
```

- Core

```
<spring:argument></spring:argument>
```

```
<spring:bind path=""></spring:bind>
```

```
<spring:escapeBody></spring:escapeBody>
```

```
<spring:eval expression=""></spring:eval>
```

```
<spring:hasBindErrors name=""></spring:hasBindErrors>
```

```
<spring:htmlEscape defaultHtmlEscape=""></spring:htmlEscape>
```

```
<spring:message></spring:message>
```

```
<spring:nestedPath path=""></spring:nestedPath>
```

```
<spring:param name=""></spring:param>
```

```
<spring:theme></spring:theme>
```

```
<spring:transform value=""></spring:transform>
```

```
<spring:url value=""></spring:url>
```

Paths Absolutos

En ocasiones, se requiere acceder a un controlador desde distintas JSP, las cuales estan a distinto nivel en el path, por ejemplo desde /**gestion**/**persona** y desde /**administracion**, se quiere acceder a /**buscar**, teniendo en cuenta que la propiedad **action** representa un path relativo, no serviria en mismo formulario, salvo que se pongan path absolutos, para los cual, se necesita obtener la url de la aplicación, hay varias alternativas

- Expresiones EL

```
<form action="${pageContext.request.contextPath}/buscar" method="GET" />
```

- Libreria de etquetas JSTL core

```
<form action="<c:url value="/buscar" />" method="GET" />
```

Inicialización

Otra opción para inicializar los objetos necesarios para el formulario, sería crear un método anotado con **@ModelAttribute**, indicando la clave del objeto del Modelo que disparará la ejecución de este método, dado que por defecto un objeto definido como **ModelAttribute** se situa en **HttpServletRequest** que es donde se ira a buscar al renderizar la JSP del formulario.

```
@ModelAttribute("persona")
public Persona initPersona(){
    return new Persona();
}
```

Validaciones

Spring MVC soporta validaciones de JSR-303.

Para aplicarlas se necesita una implementación como **hibernate-validator**, para añadirla con Maven.

```
<dependency>
    <groupId>org.hibernate</groupId>
    <artifactId>hibernate-validator</artifactId>
    <version>5.1.3.Final</version>
</dependency>
```

Para activar la validación entre **View** y **Controller**, se añade a los parámetros de los métodos del **Controller**, la anotación **@Valid**.

```
@RequestMapping(method = RequestMethod.POST)
public Persona altaPersona(@Valid @RequestBody Persona persona) {}
```

Si además se quiere conocer el estado de la validación para ejecutar la lógica del controlador, se puede indicar en los parámetros que se recibe un objeto **Errors**, que tiene un método **hasErrors()** que indica si hay errores de validación.

```
public String altaPersona(@Valid @ModelAttribute("persona") Persona p,
    Errors errors, Model model){}

    if (errors.hasErrors()) {
        return "error";
    } else {
        return "ok";
    }
}
```

Y en la clase del **Model**, las anotaciones correspondientes de JSR-303

```
public class Persona {
    @NotEmpty(message="Hay que rellenar el campo nombre")
    private String nombre;
    @NotEmpty
    private String apellido;
    private int edad;
}
```

Mensajes personalizados

Como se ve en el anterior ejemplo, se ha personalizado el mensaje para la validación **@NotEmpty** del campo **nombre**

Se puede definir el mensaje en un properties, teniendo en cuenta que el property tendra la siguiente firma

```
<validador>.<entidad>.<caracteristica>
```

Por ejemplo para la validación anterior de **nombre**

```
notempty.persona.nombre = Hay que rellenar el campo nombre
```

Tambien se puede referenciar a una propiedad cualquiera, pudiendo ser cualquier clave.

```
@NotEmpty(message="{notempty.persona.nombre}")
private String nombre;
```

Anotaciones JSR-303

Las anotaciones están definidas en el paquete **javax.validation.constraints**.

- **@Max**
- **@Min**
- **@NotNull**
- **@Null**
- **@Future**
- **@Past**
- **@Size**
- **@Pattern**

Validaciones Custom

Se pueden definir validadores nuevos e incluirlos en la validación automatizada, para ello hay que implementar la interface **org.springframework.validation.Validator**

```
public class PersonaValidator implements Validator {
    @Override
    public boolean supports(Class<?> clazz) {
        return Persona.class.equals(clazz);
    }
    @Override
    public void validate(Object obj, Errors e) {
        Persona persona = (Persona) obj;
        e.rejectValue("nombre", "formulario.persona.error.nombre");
    }
}
```

NOTE	El metodo de supports, indica que clases se soportan para esta validación, si retornase true, aceptaria todas, no es lo habitual ya que tendrá al menos una caracteristica concreta que será la validada.

Una vez definido el validador, para añadirlo al flujo de validación de un **Controller**, se ha de añadir una instancia de ese validador al **Binder** del **Controller**, creando un método en el **Controller**, anotado con **@InitBinder**

```
@InitBinder
protected void initBinder(final WebDataBinder binder) {
    binder.addValidators(new PersonaValidator());
}
```

Los errores asociados a estas validaciones pueden ser visualizados en la **View** empleando la etiqueta <**form:errors**/>

```
<form:errors path="*"/>
```

NOTE	La propiedad path, es el camino que hay que seguir en el objeto de **Model** para acceder a la propiedad validada.

Internacionalización - i18n

Para poder aplicar la internacionalización, hay que trabajar con ficheros properties manejados como **Bundles**, esto en Spring se consigue definiendo un **Bean** con id **messageSource** de tipo **AbstractMessageSource**

```
<bean id="messageSource" class=
"org.springframework.context.support.ReloadableResourceBundleMessageSource">
    <property name="basename" value="/WEB-INF/messages/messages" />
</bean>
```

Una vez defido el Bean deberán existir tantos ficheros como idiomas soportados con la firma

```
/WEB-INF/messages/messages_<COD-PAIS>_<COD-DIALECTO>.properties
```

Como por ejemplo

```
/WEB-INF/messages/messages_es.properties
/WEB-INF/messages/messages_es_es.properties
/WEB-INF/messages/messages_en.properties
```

Para acceder a estos mensajes desde las **View** existe una libreria de etiquetas

```
<%@ taglib uri="http://www.springframework.org/tags" prefix="spring"%>
```

Que porporciona la etiqueta

```
<spring:message code="<clave en el properties>"/>
```

Tambien es posible emplear JSTL

```
<dependency>
    <groupId>jstl</groupId>
    <artifactId>jstl</artifactId>
    <version>1.2</version>
</dependency>
```

```
<%@ taglib prefix="fmt" uri="http://java.sun.com/jsp/jstl/fmt" %>
```

```
<fmt:message key="<clave en el properties>"/>
```

Interceptor

Permiten interceptar las peticiones al **DispatcherServlet**.

Son clases que extienden de **HandlerInterceptorAdapter**, que permite actuar sobre la petición con tres métodos.

- **preHandle()** –> Se invoca antes que se ejecute la petición, retorna un booleano, si es **True** continua la ejecución normalmente, si es **False** la para.
- **postHandle()** –> Se invoca despues de que se ejecute la petición, permite manipular el objeto **ModelAndView** antes de pasarselo a la **View**.
- **afterCompletion()** –> Called after the complete request has finished. Seldom use, cant find any use case.

Los **Interceptor** pueden ser asociados

- A cada **HandlerMapping** en particular, con la propiedad **interceptors**.

```
<bean class="org.springframework.web.servlet.handler.SimpleUrlHandlerMapping">
    <property name="mappings">
        <props>
            <prop key="/index.html">indexController</prop>
        </props>
    </property>
    <property name="interceptors">
        <list>
            <ref bean="auditoriaInterceptor" />
        </list>
    </property>
</bean>

<bean id="auditoriaInterceptor" class="com.ejemplo.mvc.interceptor.AuditoriaInterceptor"
/>

<bean id="indexController" class="com.ejemplo.mvc.interceptor.IndexController" />
```

- O de forma general a todos

Con XML, se emplearia la etiqueta del namespace **mvc**

```
<mvc:interceptors>
    <bean class="com.ejemplo.mvc.interceptor.AuditoriaInterceptor" />
</mvc:interceptors>
```

Con JavaConfig, sobreescribiendo el método **addInterceptors** obtenido por la herencia de **WebMvcConfigurerAdapter**

```
@EnableWebMvc
@Configuration
public class WebConfig extends WebMvcConfigurerAdapter {

    @Override
    public void addInterceptors(InterceptorRegistry registry) {
        registry.addInterceptor(new LocaleInterceptor());
    }

}
```

Se proporcionan las siguientes implementaciones

- **ConversionServiceExposingInterceptor** → Situa el **ConversionService** en la **request**.

LocaleChangeInterceptor → Permite interpretar el parámetro **locale** de la petición para cambiar el **Locale** de la aplicación.

- **ResourceUrlProviderExposingInterceptor** → Situa el **ResourceUrlProvider** en la **request**.
- **ThemeChangeInterceptor** → Permite interpretar el parámetro **theme** de la petición para cambiar el **Tema** (conjunto de estilos) de la aplicación.
- **UriTemplateVariablesHandlerInterceptor** → Se encarga de resolver las variables del Path y ponerlas en la **request**.
- **UserRoleAuthorizationInterceptor** → Compreba la autorizacion del usuario actual, validando sus roles.

LocaleChangeInterceptor

Se declara el **Interceptor**.

```
<mvc:interceptors>
    <bean id="localeChangeInterceptor" class=
"org.springframework.web.servlet.i18n.LocaleChangeInterceptor">
        <property name="paramName" value="language" />
    </bean>
</mvc:interceptors>
```

Para cambiar el **Locale** basta con acceder a la URL

```
http://...................?language=es
```

NOTE	Por defecto el parametro que representa el codigo idiomatico es **locale**

Se puede configurar como se almacena la referencia al **Locale**, para ello basta con definir un Bean llamado **localeResolver** de tipo

- Para almacenamiento en una **Cookie**

```
<bean id="localeResolver" class=
"org.springframework.web.servlet.i18n.CookieLocaleResolver">
    <property name="defaultLocale" value="es" />
    <property name="cookieName" value="myAppLocaleCookie"></property>
    <property name="cookieMaxAge" value="3600"></property>
</bean>
```

- Para almacenamiento en la **Session**
-

```
<bean id="localeResolver" class=
"org.springframework.web.servlet.i18n.SessionLocaleResolver" />
```

- El por defecto, busca en la cabecera **accept-language**

```
<bean id="localeResolver" class=
"org.springframework.web.servlet.i18n.AcceptHeaderLocaleResolver"/>
```

ThemeChangeInterceptor

Se declara el **Interceptor**.

```
<mvc:interceptors>
    <bean id="themeChangeInterceptor" class=
"org.springframework.web.servlet.theme.ThemeChangeInterceptor">
        <property name="paramName" value="theme" />
    </bean>
</mvc:interceptors>
```

Para cambiar el **Tema** basta con acceder a la URL

```
http://...................?theme=aqua
```

Tambien se ha de declarar un Bean que indique el nombre del fichero **properties** que almacenará el nombre de los ficheros de estilos a emplear en cada **Tema**, este Bean se ha de llamar **themeSource**

```
<bean id="themeSource" class=
"org.springframework.ui.context.support.ResourceBundleThemeSource">
    <property name="basenamePrefix" value="theme-" />
</bean>
```

Se puede configurar como se almacena la referencia al **Tema**, para ello basta con definir un Bean llamado **themeResolver** de tipo

- Para almacenamiento en una **Cookie**

```
<bean id="themeResolver" class="
org.springframework.web.servlet.theme.CookieThemeResolver">
    <property name="defaultThemeName" value="default" />
</bean>
```

- Para almacenamiento en la **Session**

```
<bean id="themeResolver" class="
org.springframework.web.servlet.i18n.SessionThemeResolver" >
    <property name="defaultThemeName" value="default" />
</bean>
```

Para poder aplicar alguna de las hojas de estilos definidas en el tema, se puede emplear la etiqueta **spring:theme**

```
<link  rel="stylesheet" href="<spring:theme code='css'/>" type="text/css" />

<spring:theme code="welcome.message" />
```

Thymeleaf

Motor de plantillas.

Define el espacio de nombres **th** que proporciona atributos para instrumentalizar las etiquetas **xhtml**.

```
<html xmlns:th="http://www.thymeleaf.org"></html>
```

Para emplearlo, se han de añadir los siguientes Bean a la configuracion

```
<bean id="templateResolver" class=
"org.thymeleaf.templateresolver.ServletContextTemplateResolver">
    <property name="prefix" value="/WEB-INF/templates/" />
    <property name="suffix" value=".html" />
    <property name="templateMode" value="HTML5" />
</bean>

<bean id="templateEngine" class="org.thymeleaf.spring4.SpringTemplateEngine">
    <property name="templateResolver" ref="templateResolver" />
</bean>

<bean class="org.thymeleaf.spring4.view.ThymeleafViewResolver">
    <property name="templateEngine" ref="templateEngine" />
</bean>
```

Con esto se considera que cualquier fichero con extension **.html** que se encuentre en la carpeta **/WEB-INF/templates/** a la que se haga referencia por el nombre del fichero como plantilla para una **View**, se resolverá con **Thymeleaf**.

Se pueden emplear dos tipos de expresiones dentro de los **HTML**, **${}** y **#...**

En Spring Boot se genera una cache para las platillas, la cual se puede deshabilitar para desarrollo

```
spring:
    thymeleaf:
        cache: false
```

HttpMessageConverters

Son los encargados de realizar el Marshall y el Unmarshall de tipologias complejas a formatos de representacion como json o xml.

El contexto de Spring los emplea cuando los métodos de los controladores emplean

- **@ResponseBody** → Indica que se debe transformar un objeto retornado por el método de controlador a un formato de representación marcado por la cabecera **Accept** y retornanrlo en el cuerpo de la respuesta.
- **@RequestBody** → Indica que se debe leer el cuerpo de la peticion como un objeto cuyo tipo de representacion viene marcado por la cabecera **ContentType**.

El uso de los converters se activa en los xml con

```
<mvc:annotation-driven/>
```

y con java config con

```
@EnableWebMvc
```

Pila por defecto de HttpMessageConverters

Por defecto al activar Spring MVC, se carga la siguiente pila de converters.

- **ByteArrayHttpMessageConverter** → convierte los arrays de bytes
- **StringHttpMessageConverter** → convierte las cadenas de caracteres
- **ResourceHttpMessageConverter** → convierte a objetos **org.springframework.core.io.Resource** desde y hacia cualquier **Stream**.
- **SourceHttpMessageConverter** → convierte a **javax.xml.transform.Source**
- **FormHttpMessageConverter** → convierte datos de formulario (application/x-www-form-urlencoded) desde y hacia un **MultiValueMap<String, String>**.

- **Jaxb2RootElementHttpMessageConverter** → convierte objetos Java desde y hacia XML, con media type **text/xml** o **application/xml** (solo sí la librería de JAXB2 está presente en el classpath).

```
<dependency>
    <groupId>com.fasterxml.jackson.dataformat</groupId>
    <artifactId>jackson-dataformat-xml</artifactId>
    <version>2.5.3</version>
</dependency>
```

- **MappingJackson2HttpMessageConverter** → convierte objetos Java desde y hacia JSON (solo sí la librería de Jackson2 está presente en el classpath).

```
<dependency>
    <groupId>com.fasterxml.jackson.core</groupId>
    <artifactId>jackson-databind</artifactId>
    <version>2.5.3</version>
</dependency>
```

- **MappingJacksonHttpMessageConverter** → convierte objetos Java desde y hacia JSON (sólo si la librería de Jackson está presente en el classpath).
- **AtomFeedHttpMessageConverter** → convierte objetos Java del tipo **Feed** que proporciona la librería Rome desde y hacia feeds Atom, media type **application/atom+xml** (solo si si la librería Roma está presente en el classpath).
- **RssChannelHttpMessageConverter** → convierte objetos Java del tipo **Channel** que proporciona la librería Rome desde y hacia feeds RSS (sólo si si la librería Roma está presente en el classpath).

Personalizacion de la Pila de HttpMessageConverters

La Pila generada por defecto se puede modificar, para ello en XML se hace

```
<mvc:annotation-driven>
    <mvc:message-converters>
        <bean class=
"org.springframework.http.converter.json.MappingJackson2HttpMessageConverter"/>
    </mvc:message-converters>
</mvc:annotation-driven>
```

Y con Javaconfig

```
public class WebConfig extends WebMvcConfigurerAdapter {
    @Override
    public void configureMessageConverters(List<HttpMessageConverter<?>> converters) {
        messageConverters.add(new MappingJackson2HttpMessageConverter());
        super.configureMessageConverters(converters);
    }
}
```

La clase **RestTemplate** tambien emplea los **HttpMessageConverter** para realizar los marshall, pudiendo establecer la pila de la siguiente manera

```
RestTemplate restTemplate = new RestTemplate();
restTemplate.setMessageConverters(getMessageConverters());
```

Rest

Los servicios REST son servicios basados en recursos, montados sobre HTTP, donde se da significado al Method HTTP.

La palabra REST viene de

- **Representacion**: Permite representar los recursos en multiples formatos, aunque el mas habitual es JSON.
- **Estado**: Se centra en el estado del recurso y no en las operaciones que se pueden realizar con el.
- **Transferencia**: Transfiere los recursos al cliente.

Los significados que se dan a los Method HTTP son:

- **POST**: Permite crear un nuevo recurso.
- **GET**: Permite leer/obtener un recurso existente.
- **PUT** o **PATCH**: Permiten actualizar un recurso existente.
- **DELETE**: Permite borrar un recurso.

Spring MVC, ofrece una anotacion **@RestController**, que auna las anotaciones **@Controller** y **@ResponseBody**, esta ultima empleada para representar la respuesta directamente con los objetos retornados por los métodos de controlador.

```
@RestController
@RequestMapping(path="/personas")
public class ServicioRestPersonaControlador {

    @RequestMapping(path="/{id}", method= RequestMethod.GET, produces=MediaType
.APPLICATION_JSON_VALUE)
    public Persona getPersona(@PathVariable("id") int id){
        return new Persona(1, "victor", "herrero", 37, "M", 1.85);
    }
}
```

De esta representación se encargan los **HttpMessageConverter**.

Personalizar el Mapping de la entidad

En transformaciones a XML o JSON, de querer personalizar el Mapping de la entidad retornada, se puede hacer empleando las anotaciones de JAXB, como son **@XmlRootElement**, **@XmlElement** o **@XmlAttribute**.

Estado de la petición

Cuando se habla de servicios REST, es importante ofrecer el estado de la petición al cliente, para ello se emplea el codigo de estado de HTTP.

Para incluir este codigo en las respuestas, se puede encapsular las entidades retornadas con **ResponseEntity**, el cual es capaz de representar tambien el codigo de estado con las constantes de **HttpStatus**

```
@RequestMapping(value="/{id}", method=RequestMethod.GET)
public ResponseEntity<Spittle> spittleById(@PathVariable long id) {
    Spittle spittle = spittleRepository.findOne(id);
    HttpStatus status = spittle != null ? HttpStatus.OK : HttpStatus.NOT_FOUND;
    return new ResponseEntity<Spittle>(spittle, status);
}
```

Localización del recurso

En la creación del recurso, peticion POST, se ha de retornar en la cabera **location** de la respuesta la Url para acceder al recurso que se acaba de generar, siendo estas cabeceras retornadas gracias de nuevo al objeto **ResponseEntity**

```
HttpHeaders headers = new HttpHeaders();
URI locationUri = URI.create("http://localhost:8080/spittr/spittles/" + spittle.getId());
headers.setLocation(locationUri);
ResponseEntity<Spittle> responseEntity = new ResponseEntity<Spittle>(spittle, headers,
HttpStatus.CREATED)
```

Cliente se servicios con RestTemplate

Las operaciones que se pueden realizar con RestTemplate son

- **Delete** → Realiza una petición DELETE HTTP en un recurso en una URL especificada

```
public void deleteSpittle(long id) {
    RestTemplate rest = new RestTemplate();
    rest.delete(URI.create("http://localhost:8080/spittr-api/spittles/" + id));
}
```

- **Exchange** → Ejecuta un método HTTP especificado contra una URL, devolviendo un ResponseEntity que contiene un objeto mapeado del cuerpo de respuesta
- **Execute** → Ejecuta un método HTTP especificado contra una URL, devolviendo un objeto mapeado en el cuerpo de la respuesta.
- **GetForEntity** → Envía una solicitud HTTP GET, devolviendo un ResponseEntity que contiene un objeto mapeado del cuerpo de respuesta

```
public Spittle fetchSpittle(long id) {
    RestTemplate rest = new RestTemplate();
    ResponseEntity<Spittle> response = rest.getForEntity("http://localhost:8080/spittr-
api/spittles/{id}", Spittle.class, id);
    if(response.getStatusCode() == HttpStatus.NOT_MODIFIED) {
        throw new NotModifiedException();
    }
        return response.getBody();
    }
}
```

- **GetForObject** → Envía una solicitud HTTP GET, devolviendo un objeto asignado desde un cuerpo de respuesta

```
public Spittle[] fetchFacebookProfile(String id) {
    Map<String, String> urlVariables = new HashMap<String, String>();
    urlVariables.put("id", id);
    RestTemplate rest = new RestTemplate();
    return rest.getForObject("http://graph.facebook.com/{spitter}", Profile.class,
urlVariables);
}
```

- **HeadForHeaders** → Envía una solicitud HTTP HEAD, devolviendo los encabezados HTTP para los URL de recursos
- **OptionsForAllow** → Envía una solicitud HTTP OPTIONS, devolviendo el encabezado Allow URL especificada
- **PostForEntity** → Envia datos en el cuerpo de una URL, devolviendo una ResponseEntity que contiene un objeto en el cuerpo de respuesta

```
    RestTemplate rest = new RestTemplate();
    ResponseEntity<Spitter> response = rest.postForEntity("http://localhost:8080/spittr-
api/spitters", spitter, Spitter.class);
    Spitter spitter = response.getBody();
    URI url = response.getHeaders().getLocation();
}
```

- **PostForLocation** → POSTA datos en una URL, devolviendo la URL del recurso recién creado

```
public String postSpitter(Spitter spitter) {
    RestTemplate rest = new RestTemplate();
    return rest.postForLocation("http://localhost:8080/spittr-api/spitters", spitter)
.toString();
}
```

- **PostForObject** → POSTA datos en una URL, devolviendo un objeto mapeado de la respuesta cuerpo

```
public Spitter postSpitterForObject(Spitter spitter) {
    RestTemplate rest = new RestTemplate();
    return rest.postForObject("http://localhost:8080/spittr-api/spitters", spitter,
Spitter.class);
}
```

- **Put** → PUT pone los datos del recurso en la URL especificada

```
public void updateSpittle(Spittle spittle) throws SpitterException {
    RestTemplate rest = new RestTemplate();
    String url = "http://localhost:8080/spittr-api/spittles/" + spittle.getId();
    rest.put(URI.create(url), spittle);
}
```

Spring Test

Framework, que proporciona un runner para poder ejecutar Test que carguen un contexto de spring

La dependencias de Maven

```
<dependency>
    <groupId>org.springframework</groupId>
    <artifactId>spring-test</artifactId>
    <version>4.3.5.RELEASE</version>
</dependency>
```

Para emplearlo, se han de anotar las clases de Test con

```
@RunWith(SpringJUnit4ClassRunner.class)
@ContextConfiguration(classes=Configuracion.class)
```

Donde el la clase **Configuracion** se definirá el contexto de spring con los beans a probar, de ser una prueba unitaria, solo necesitaremos definir el **SUT**.

Al test se le inyectara el **SUT** con **@Autowired**

```
@Autowired
private Servicio sut;
```

Mocks

Si es una prueba unitaria sobre un componente que tiene dependencias, será necesario cubrir esas dependencias con objetos **Mock** que describan la funcionalidad esperada por el componente a probrar en el entorno de pruebas, para ello se puede emplear por ejemplo **Mockito**

La dependencias de Maven

```
<dependency>
    <groupId>org.mockito</groupId>
    <artifactId>mockito-all</artifactId>
    <version>1.10.19</version>
</dependency>
```

Este framework, permite definir objetos **Mock** con la anotacion **@Mock**

```
@RunWith(SpringJUnit4ClassRunner.class)
@ContextConfiguration(classes = Configuracion.class)
public class TestCliente {
    @Mock
    private Pedido pedido;
}
```

La cual se activa bien ejecutando con el Runner de Mockito **MockitoJUnitRunner** o bien ejecutando previamente a los Test lo siguiente

```
@Before
public void setup() {
    MockitoAnnotations.initMocks(this);
}
```

Este último será el caso para **Spring-Test**, ya que el Runner debe ser el de Spring.

Para poder emplear el **Mock**, habrá que relacionar los Bean, cosa que todavia no ha ocurrido, ya que cada uno lo genera un contenedor, para ello, se ha de indicar al **SUT**, que reciba las dependencias de **Mocks** de Mockito, para ello se ha de anotar con **@InjectMocks**

```
@RunWith(SpringJUnit4ClassRunner.class)
@ContextConfiguration(classes = Configuracion.class)
public class TestCliente {

    @Mock
    private Pedido pedido;

    @Autowired
    @InjectMocks
    private Cliente cliente; //La tipologia Cliente tiene una dependencia con un bean
Pedido
}
```

Una vez establecida la dependencia, solo falta definir los comportamientos de los **Mocks** ante el

entorno de pruebas del **SUT**.

MVC Mocks

El framework proporciona una clase **MockMvc** que permite comprobar el comportamiento de los Controladores simulando su ejecución en un contenedor web.

```
Runner.class)
@ContextConfiguration(classes = Configuracion.class)
public class TestControlador {

    private MockMvc mockMvc;

    @Autowired
    private Controlador sut; //Bean de Spring de tipo @Controller

    @Before
    public void init() {
        mockMvc = MockMvcBuilders.standaloneSetup(controlador).build();
    }
}
```

Una vez inicializado el **MockMvc** y asociado al **SUT**, y dada la siguiente implementación del controlador

```
@Controller
public class Controlador {
    @RequestMapping(value = "/", method = RequestMethod.GET)
    public String metodo(@RequestParam String param, Model model) {
        List<String> models = Arrays.asList(new String[]{"dato","otro"});
        model.addAttribute("datos", models);
        return "resultado";
    }
}
```

Se relizan las pruebas, para lo cual el API ofrece métodos estaticos en las clases **org.springframework.test.web.servlet.request.MockMvcRequestBuilders** y **org.springframework.test.web.servlet.result.MockMvcResultMatchers**

```
@Test
public void codigoDeRespuestaCorrecto() throws Exception {
    mockMvc.perform(get(URL + "?param=victor")).andExpect(status().isOk());
}

@Test
public void contenidoRespuestaCorrecto() throws Exception {
    mockMvc.perform(get(URL + "?param=info")).andExpect(model().attributeExists("datos")
);
}

@Test
public void methodNoValido() throws Exception {
    mockMvc.perform(put(URL)).andExpect(status().isMethodNotAllowed());
}

@Test
public void peticionMalConstruida() throws Exception {
    mockMvc.perform(get(URL)).andExpect(status().isBadRequest());
}
```

Mockito

Para poder crear un buen conjunto de pruebas unitarias, es necesario centrarse exclusivamente en la unidad (normalmente será un metodo de una clase concreto) a testear, para ello se pueden simular, con **Mocks** el resto de clases involucradas, de esta manera se crean test unitarios potentes que permiten detectar los errores allí donde se producen y no en dependencias del supuesto código probado.

Mockito es una herramienta que permite generar **Mocks** dinámicos. Estos pueden ser de clases concretas o de interfaces. Esta parte de la generación de las pruebas, no se centra en la validación de los resultados, sino en los que han de retornar aquellos componentes de los que depende la clase probada.

La creación de pruebas con Mockito se divide en tres fases

- **Stubbing**: Definición del comportamiento de los Mock ante unos datos concretos.
- **Invocación**: Utilización de los Mock, al interaccionar la clase que se esta probando con ellos.
- **Validación**: Validación del uso de los Mock.

Se pueden definir los **Mock** con

- La anotacion @Mock aplicada sobre un atributo de clase.

```
@Mock
private IUserDAO mockUserDao;
```

De emplearse las anotaciones, se ha de ejecutar la siguiente sentencia para que se procesen dichas anotaciones y se generen los objetos **Mock**

```
MockitoAnnotations.initMocks(testClass);
```

O bien emplear un **Runner** especifico de Mockito en la clase de Test que emplee Mockito, el **MockitoJUnitRunner**

```
@RunWith(MockitoJUnitRunner.class)
```

- O con el método estatico **mock**.

```
private IDataSesionUserDAO mockDataSesionUserDao = mock(IDataSesionUserDAO.class);
```

Stubing

Se persigue definir comportamientos del **Mock**, para ello se emplean los métodos estaticos de la clase **org.mockito.Mockito**, que son

- atLeast
- atMost
- atLeastOnce
- doNothing
- doReturn
- doThrow
- when
- inOrder
- never
- only
- verify
- mock

Y de la clase **org.mockito.Matchers**, que son

- any
- anyString
- anyObject
- contains
- endsWith
- startsWith
- eq
- isA
- isNull
- isNotNull

Algunos ejemplos de definicion de comportamientos del Mock

```
when(mockUserDao.getUser(validUser.getId())).thenReturn(validUser);

when(mockUserDao.getUser(invalidUser.getId())).thenReturn(null);

when(mockDataSesionUserDao.deleteDataSesion((User) eq(null), anyString())).thenThrow(new
OperationNotSupportedException());

when(mockDataSesionUserDao.updateDataSesion(eq(validUser), eq(validId), anyObject()))
.thenReturn(true);

when(mockDataSesionUserDao.updateDataSesion(eq(validUser), eq(invalidId), anyObject()))
.thenThrow(new OperationNotSupportedException());

when(mockDataSesionUserDao.updateDataSesion((User) eq(null), anyString(), anyObject()))
.thenThrow(new OperationNotSupportedException());
```

Por defecto todos los métodos que devuelven valores de un mock devuelven null, una colección vacía o el tipo de dato primitivo apropiado, salvo que se defina un comportamiento distinto.

Verificación

Se puede verificar el orden en el que se han ejecutado los métodos del **Mock**, pudiendo llegar a diferenciar el orden de invocación de un mismo método por los parametros enviados.

En este ejemplo se esta verificando que el orden de ejecucion de los métodos **getUser** *del mock* **mockUserDao**, *se ejecuta antes que el método* **deleteDataSesion** *del mock* **mockDataSesionUserDao**

```
ordered = inOrder(mockUserDao, mockDataSesionUserDao);
ordered.verify(mockUserDao).getUser(validUser.getId());
ordered.verify(mockDataSesionUserDao).deleteDataSesion(validUser, validId);
```

Tambien se puede verificar el numero de veces que se ha invocado una funcionalidad

En este ejemplo se verifica que el método **someMethod** *del* **mock** *no se ejecuta nunca, que el método* **someMethod(int)** *se ejecuta 1 sola vez y que el método* **someMethod(string)** *se ejecuta 2 veces.*

```
verify(mock, never()).someMethod();
verify(mock, only()).someMethod(2);
verify(mock, times(2)).someMethod("some arg");
```

Log

Para configurar los Logs de una aplicación, habitualmente se emplea alguna libreria de fachada que abstrae de la implementación real a emplear como JCL (Jakarta Commons Logging) o SLF4J (Simple Logging Facade for Java)

En el caso de Spring por defecto emplea JCL, aunque lo mas habitual en la actualidad es emplearlo con SLF4J con Logback.

SLF4J (Simple Logging Facade for Java)

SLF4J permite abstraer entre los siguientes frameworks de log:

- java.util.logging
- logback
- log4j

Para incluirlo en un proyecto Spring, hay primero que excluir la dependencia que presenta Spring con JCL

```
<dependency>
    <groupId>org.springframework</groupId>
    <artifactId>spring-context</artifactId>
    <version>${spring.version}</version>
    <exclusions>
        <exclusion>
            <groupId>commons-logging</groupId>
            <artifactId>commons-logging</artifactId>
        </exclusion>
    </exclusions>
</dependency>
```

Y luego añadir una libreria que hace de puente entre JCL y SLF4J.

```
<dependency>
    <groupId>org.slf4j</groupId>
    <artifactId>jcl-over-slf4j</artifactId>
    <version>1.7.0</version>
</dependency>
```

Para finalmente incluir la dependencia con SLF4J

```
<dependency>
    <groupId>org.slf4j</groupId>
    <artifactId>slf4j-api</artifactId>
    <version>1.7.0</version>
</dependency>
```

Una vez incluido, se deberá añadir la implementación que se desee emplear junto son su configuracion.

Para emplear esta fachada y conseguir desacoplarse del framework de logging, se debe emplear el API de SLF4J

```
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;

...

private static final Logger logger = LoggerFactory.getLogger(MiClase.class);

...

logger.debug();
```

SLF4J + Logback

Logback, es el sucesor de Logj4.

Para incluir como framework de logging a **Logback**, habrá que añadir la siguiente dependencia al classpath

```
<dependency>
    <groupId>ch.qos.logback</groupId>
    <artifactId>logback-classic</artifactId>
    <version>${logback.version}</version>
</dependency>
```

Este framework, se configura con un fichero **logback.xml** en la raiz del classpath, no hay un **XSD** (schema xml) oficial para la implmentacion de dicho fichero, pero se pueden encontrar versiones no oficiales pero utiles como la siguiente

```
<configuration xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:noNamespaceSchemaLocation="http://www.padual.com/java/logback.xsd">
</configuration>
```

En este fichero, el nodo principal es <**configuration**> y dentro de el, se definirán

- **Appenders** → Basados en la interface **ch.qos.logback.core.Appender**, permiten configurar destinos del log, existen muchas implementaciones: a fichero, consola, BBDD, JMS, Socket, SMTP, ... cada una con sus propias configuraciones.
- **Encoder** → Permiten definir los patrones de pintado del log, para ello emplean **Layouts**, donde uno de los más habituales es el **PatternLayout**, que permite definir una patrón que accede a variables predefinidas en el contexto del logger como la hora, el thread, el nivel, la clase que ejecuta la sentencia de logging, ...
- **Filter** → Permite indicar bajo que condiciones el **Appender** realizará su trabajo.

- **Logger** → Permite definir **Appenders** sobre un paquete del classpath particular.
- **Root** → Permite definir los **Appenders** del log general.

Un ejemplo de configuracion seria.

```
<?xml version="1.0" encoding="UTF-8"?>
    <configuration>
        <appender name="consoleAppender" class="ch.qos.logback.core.ConsoleAppender">
            <encoder>
                <Pattern>%d{HH:mm:ss.SSS} [%thread] %-5level %logger{36} - %msg %n
                </Pattern>
            </encoder>
            <filter class="ch.qos.logback.classic.filter.ThresholdFilter">
                <level>TRACE</level>
            </filter>
        </appender>

        <appender name="dailyRollingFileAppender" class=
"ch.qos.logback.core.rolling.RollingFileAppender">
            <File>c:/tmp/rest-demo.log</File>
            <rollingPolicy class="ch.qos.logback.core.rolling.TimeBasedRollingPolicy">
                <!-- daily rollover -->
                <FileNamePattern>rest-demo.%d{yyyy-MM-dd}.log</FileNamePattern>

                <!-- keep 30 days' worth of history -->
                <maxHistory>30</maxHistory>
            </rollingPolicy>

            <encoder>
                <Pattern>%d{HH:mm:ss.SSS} [%thread] %-5level %logger{35} - %msg
%n</Pattern>
            </encoder>
        </appender>
        <appender name="minuteRollingFileAppender" class=
"ch.qos.logback.core.rolling.RollingFileAppender">
            <rollingPolicy class="ch.qos.logback.core.rolling.TimeBasedRollingPolicy">
                <!-- rollover every minute -->
                <FileNamePattern>c:/tmp/minutes/rest-demo-minute.%d{yyyy-MM-dd_HH-
mm}.log</FileNamePattern>

                <!-- keep 30 minutes' worth of history -->
                <maxHistory>30</maxHistory>
            </rollingPolicy>

            <encoder>
                <Pattern>%-4relative [%thread] %-5level %logger{35} - %msg %n</Pattern>
            </encoder>
```

```
        </appender>

        <logger name="com.ejemplo.spring.log" additivity="false">
            <level value="DEBUG" />
            <appender-ref ref="dailyRollingFileAppender"/>
            <appender-ref ref="minuteRollingFileAppender"/>
            <appender-ref ref="consoleAppender" />
        </logger>

        <root>
            <level value="INFO" />
            <appender-ref ref="consoleAppender" />
        </root>
    </configuration>
```

SLF4J + Log4j

Para incluir como framework de logging a **Log4j**, habrá que añadir la siguiente dependencia al classpath.

```
<dependency>
  <groupId>org.slf4j</groupId>
  <artifactId>slf4j-log4j12</artifactId>
  <version>1.8.0-beta0</version>
</dependency>
```

Y posteriormente configurar log4j con el fichero **log4j.properties** en la raiz del classpath

```
# Root logger option
log4j.rootLogger=INFO, file, stdout

# Direct log messages to a log file
log4j.appender.file=org.apache.log4j.RollingFileAppender
log4j.appender.file.File=C:\\logging.log
log4j.appender.file.MaxFileSize=10MB
log4j.appender.file.MaxBackupIndex=10
log4j.appender.file.layout=org.apache.log4j.PatternLayout
log4j.appender.file.layout.ConversionPattern=%d{yyyy-MM-dd HH:mm:ss} %-5p %c{1}:%L - %m%n

# Direct log messages to stdout
log4j.appender.stdout=org.apache.log4j.ConsoleAppender
log4j.appender.stdout.Target=System.out
log4j.appender.stdout.layout=org.apache.log4j.PatternLayout
log4j.appender.stdout.layout.ConversionPattern=%d{yyyy-MM-dd HH:mm:ss} %-5p %c{1}:%L -
%m%n
```

SLF4J + JUL

Para incluir como framework de logging a **java.util.logging**, habrá que añadir la siguiente dependencia al classpath.

```
<dependency>
  <groupId>org.slf4j</groupId>
  <artifactId>slf4j-jdk14</artifactId>
  <version>1.8.0-beta0</version>
</dependency>
```

Y configurarlo con el fichero **logging.properties** en la raiz del classpath.

```
handlers= java.util.logging.FileHandler, java.util.logging.ConsoleHandler

# Default global logging level.
# This specifies which kinds of events are logged across
# all loggers. For any given facility this global level
# can be overriden by a facility-specific level.
# Note that the ConsoleHandler also has a separate level
# setting to limit messages printed to the console.

.level= INFO

# Limit the messages that are printed on the console to INFO and above.

java.util.logging.ConsoleHandler.level = INFO
java.util.logging.ConsoleHandler.formatter = java.util.logging.SimpleFormatter

# default file output is in user's home directory.

java.util.logging.FileHandler.pattern = %h/java%u.log
java.util.logging.FileHandler.limit = 50000
java.util.logging.FileHandler.count = 1
java.util.logging.FileHandler.formatter = java.util.logging.XMLFormatter

# The logger with javax.jms.connection name space will write
# Level.INFO messages to its output handler(s). In this configuration
# the ouput handler is set to java.util.logging.ConsoleHandler.

com.ejemplo.spring.level = INFO
```