Spring

Victor Herrero Cazurro

Contenidos

¿Que es Spring?

Framework para el de desarrollo de aplicaciones java.

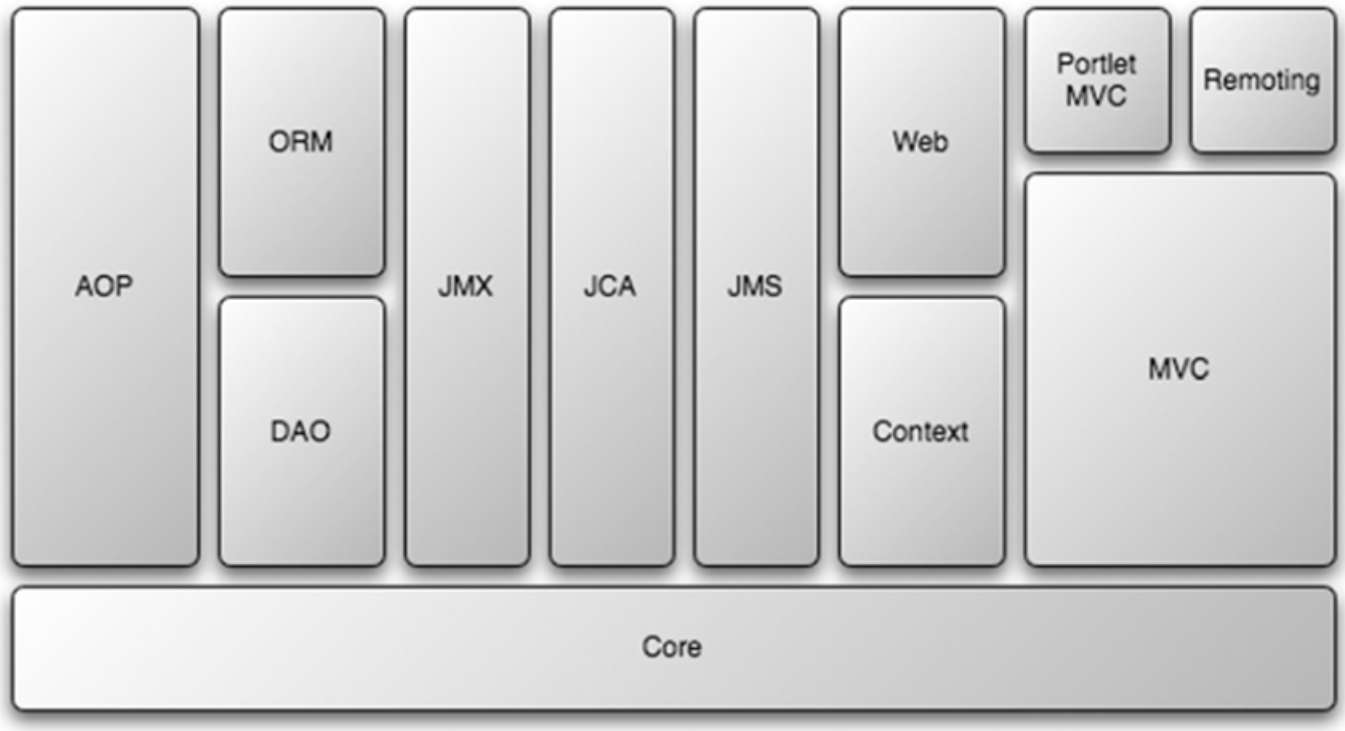
Es un contenedor ligero de POJOS que se encarga de la creación de beans (Factoría de Beans) mediante la Inversión de control y la inyección de dependencias.

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 Boot

Introduccion

Framework orientado a la construcción/configuración de proyectos de la familia Spring basado en **Convention-Over-Configuration**, por lo que minimiza la cantidad de código de configuración de las aplicaciones.

Afecta principalmente a dos aspectos de los proyectos

- **Configuracion de dependencias**: Proporcionado por **Starters** Aunque sigue empleando Maven o Gradle para configurar las dependencias del proyecto, abstrae de las versiones de los APIs y lo que es más importante de las versiones compatibles de unos APIs con otros, dado que proporciona un conjunto de librerias que ya están probadas trabajado juntas.
- **Configuracion de los APIs**: Cada API de Spring que se incluye, ya tendrá una preconfiguración por defecto, la cual si se desea se podrá cambiar, además de incluir elementos tan comunes en los desarrollos como un contenedor de servlets embebido ya configurado, estas preconfiguraciones se establecen simplemente por el hecho de que la libreria esté en el classpath, como un Datasource de una base de datos, JDBCTemplate, Java Persistence API (JPA), Thymeleaf templates, Spring Security o Spring MVC.

Además proporciona otras herramientas como

- La consola Spring Boot CLI
- Actuator

Instalación de Spring Boot CLI

Permite la creación de aplicaciones Spring, de forma poco convencional, centrandose unicamente en el código, la consola se encarga de resolver dependencias y configurar el entorno de ejecución.

Emplea scripts de Groovy.

Para descargar la distribución pinchar aquí

Descomprimir y añadir a la variable entorno PATH la ruta **$SPRING_BOOT_CLI_HOME/bin**

Se puede acceder a la consola en modo ayuda (completion), con lo que se obtiene ayuda para escribir los comandos con TAB, para ello se introduce

```
> spring shell
```

Una vez en la consola se puede acceder a varios comandos uno de ellos es el de la ayuda general **help**

```
Spring-CLI# help
```

O la ayuda de alguno de los comandos

```
Spring-CLI# help init
```

Con Spring Boot se puede creear un poryecto MVC tan rapido como definir la siguiente clase Groovy **HelloController.groovy**

```
@RestController
class HelloController {

    @RequestMapping("/")
    def hello() {
        return "Hello World"
    }

}
```

Y ejecutar desde la consola Spring Boot CLI

```
> spring run HelloController.groovy
```

La consola se encarga de resolver las dependencias, de compilar y de establecer las configuraciones por defecto para una aplicacion Web MVC, en el web.xml, ... por lo que una vez ejecutado el comando de la consola, al abrir el navegador con la url http://localhost:8080 se accede a la aplicación.

Si se dispone de mas de un fichero **groovy**, se puede lanzar todos los que se quiera con el comando

```
> spring run *.groovy
```

El directorio sobre el que se ejecuta el comando es considerado el root del classpath, por lo que si se añade un fichero **application.properties**, este permite configurar el proyecto.

Si se quiere añadir motores de plantillas, se deberá incluir la dependencia, lo cual se puede hacer con **Grab**, por ejemplo para añadir **Thymeleaf**

```
@Grab(group='org.springframework.boot', module='spring-boot-starter-thymeleaf', version=
'1.5.7.RELEASE')

@Controller
class Application {
    @RequestMapping("/")
    public String greeting() {
        return "greeting"
    }
}
```

Y definir las plantillas en la carpeta **templates**, en este caso **templates/greeting.html**

Si se desea contenido estatico, este se debe poner en la carpeta **resources** o **static**

Creación e implementación de una aplicación

Lo primero a resolver al crear una aplicación son las dependencias, para ellos Spring Boot ofrece el siguiente mecanismo basando en la herencia del POM.

```
<project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi=
"http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
http://maven.apache.org/xsd/maven-4.0.0.xsd">

    ...

    <parent>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-parent</artifactId>
        <version>1.4.2.RELEASE</version>
        <relativePath/> <!-- lookup parent from repository -->
    </parent>

    ...

</project>
```

De no poderse establecer dicha herencia, por heredar de otro proyecto, se ofrece la posibilidad de añadir la siguiente dependedencia.

```
<project>

    ...

    <dependencyManagement>
         <dependencies>
            <dependency>
                <!-- Import dependency management from Spring Boot -->
                <groupId>org.springframework.boot</groupId>
                <artifactId>spring-boot-dependencies</artifactId>
                <version>1.4.2.RELEASE</version>
                <type>pom</type>
                <scope>import</scope>
            </dependency>
        </dependencies>
    </dependencyManagement>

    ...

</project>
```

Esta dependencia permite a Spring Boot hacer el trabajo sucio para manejar el ciclo de vida de un proyecto Spring normal, pero normalmente se precisarán otras dependencias, para esto Spring Boot ofrece los **Starters**, por ejemplo esta seria la dependencia para un proyecto Web MVC

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

Una vez solventadas las dependencias, habrá que configurar el proyecto, ya hemos mencionado que la configuración quedará muy reducida, en este caso unicamnete necesitamos definir una clase anotada con **@SpringBootApplication**

```
@SpringBootApplication
public class HolaMundoApplication {
    ...
}
```

Esta anotacion en realidad es la suma de otras tres:

- @Configuration → Se designa a la clase como un posible origen de definiciones de Bean.

- @ComponentScan → Se indica que se buscarán otras clases con anotaciones que definan componentes de Spring como @Controller
- @EnableAutoConfiguration → Es la que incluye toda la configuración por defecto para los distintos APIs seleccionados.

Con esto ya se tendría el proyecto preparado para incluir unicamente el código de aplicación necesario, por ejemplo un Controller de Spring MVC

```
@Controller
public class HolaMundoController {
    @RequestMapping("/")
    @ResponseBody
    public String holaMundo() {
        return "Hola Mundo!!!!!";
    }
}
```

Una vez finalizada la aplicación, se podría ejecutar de varias formas

- Como jar autoejecutable, para lo que habrá que definir un método **Main** que invoque **SpringApplication.run()**

```
@SpringBootApplication
public class HolaMundoApplication {
    public static void main(String[] args) {
        SpringApplication.run(HolaMundoApplication.class, args);
    }
}
```

Y posteriormente ejecutandolo con

- Una tarea de Maven

```
mvn spring-boot:run
```

- Una tarea de Gradle

```
gradle bootRun
```

- O como jar autoejecutable, generando primero el jar

Con Maven

```
mvn package
```

O Gradle

```
gradle build
```

Y ejecutando desde la linea de comandos

```
java -jar HolaMundo-0.0.1-SNAPSHOT.jar
```

- O desplegando como WAR en un contenedor web, para lo cual hay que añadir el plugin de WAR
  - En Maven, con cambiar el package bastará

```
<packaging>war</packaging>
```

- En Gradle alicando el plugin de WAR y cambiando la configuracion JAR por la WAR

```
apply plugin: 'war'

war {
    baseName = 'HolaMundo'
    version = '0.0.1-SNAPSHOT'
}
```

En estos casos, dado que no se ha generado el **web.xml**, es necesario realizar dicha inicialización, para ello Srping Boot ofrece la clase **org.springframework.boot.web.support.SpringBootServletInitializer**

```
public class HolaMundoServletInitializer extends SpringBootServletInitializer {
    @Override
    protected SpringApplicationBuilder configure(SpringApplicationBuilder builder) {
        return builder.sources(HolaMundoApplication.class);
    }
}
```

Uso de plantillas

Los proyectos **Spring Boot Web** vienen configurados para emplear plantillas, basta con añadir el starter del motor deseado y definir las plantillas en la carpeta **src/main/resources/templates**.

Algunos de los motores a emplear son Thymeleaf, freemaker, velocity, jsp, ...

Thymeleaf

Motor de plantillas que se basa en la instrumentalización de **html** con atributos obtenidos del esquema **th**

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

Para añadir esta caracteristica al proyecto, se añade la dependencia de Maven

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

Por defecto cualquier **String** retornado por un **Controlador** será considerado el nombre de un **html** instrumentalizado con **thymeleaf** que se ha de encontrar en la carpeta **/src/main/resources/templates**

```
<!DOCTYPE html>
<html>
<head>
    <meta charset="ISO-8859-1"></meta>
    <title>Insert title here</title>
</head>
<body>
    <span th:text="${mensaje}"></span>
    <span th:text="#"></span>
</body>
</html>
```

NOTE No es necesario indicar el espacio de nombres en el html

JSP

Para poder emplear **JSP** en lugar de **Thymeleaf**, hay dos opciones, la primera es definir el proyecto de Spring Boot como War en el pom.xml, definiendo la siguiente configuracion en el contexto de Spring

```
@SpringBootApplication
public class SampleWebJspApplication extends SpringBootServletInitializer {

    @Override
    protected SpringApplicationBuilder configure(SpringApplicationBuilder application) {
        return application.sources(SampleWebJspApplication.class);
    }

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

}
```

y las siguientes propiedades en el fichero **application.properties**

```
spring.mvc.view.prefix: /WEB-INF/views/
spring.mvc.view.suffix: .jsp
```

NOTE El directorio desde donde creacerá **WEB-INF**, sera **src/main/webapp**

La segunda opcion, será mantener el tipo de proyecto como Jar y añadir las siguientes dependencias al **pom.xml**

```
<dependency>
    <groupId>org.apache.tomcat.embed</groupId>
    <artifactId>tomcat-embed-jasper</artifactId>
</dependency>
<dependency>
    <groupId>javax.servlet</groupId>
    <artifactId>jstl</artifactId>
</dependency>
```

Por último indicar donde encontrar los ficheros mediante las siguientes propiedades en el fichero **application.properties**

```
spring.mvc.view.prefix: /WEB-INF/views/
spring.mvc.view.suffix: .jsp
```

NOTE El directorio desde donde creacerá **WEB-INF**, sera **src/main/resources/META-INF/resources/**

Recursos estaticos

Si se desean publicar recursos estaticos (html, js, css, ...), se pueden incluir en los proyectos en las rutas:

- **src/main/resources/META-INF/resources**
- **src/main/resources/resources**
- **src/main/resources/static**
- **src/main/resources/public**

Siendo el descrito el orden de inspeccion.

Webjars

Desde hace algun tiempo se encuentran disponibles como dependencias de Maven las distribuciones de algunos frameworks javascript bajo el groupid **org.webjars**, pudiendo añadir dichas dependencias a los proyectos para poder gestionar con herramientas de construccion como Maven o Gradle tambien las versiones de los frameworks javascript.

Estos artefactos tienen incluido los ficheros js, en la carpeta **/META-INF/resources/webjars/<artifactId>/<version>**, con lo que las dependencias hacia los ficheros javascript de los framework añadidos con Maven será **webjars/<artifactId>/<version>/<artifactId>.min.js**

```
<html>
<head>
    <script src="webjars/jquery/2.0.3/jquery.min.js"></script>
...
```

Recolección de métricas

El API de Actuator, permite recoger información del contexto de Spring en ejecución, como

- Qué beans se han configurado en el contexto de Spring.
- Qué configuraciones automáticas se han establecido con Spring Boot.
- Qué variables de entorno, propiedades del sistema, argumentos de la línea de comandos están disponibles para la aplicación.
- Estado actual de los subprocesos
- Rastreo de solicitudes HTTP recientes gestionadas por la aplicación
- Métricas relacionadas con el uso de memoria, recolección de basura, solicitudes web, y uso de fuentes de datos.

Estas metricas se exponen via Web o via shell.

Para activarlo, es necesario incluir una dependencia

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

Y a partir de ahi, bajo la aplicación desplegada, se encuentran los path con la info, que retornan JSON

- Estado

http://localhost:8080/ListadoDeTareas/health

- Mapeos de URL

http://localhost:8080/ListadoDeTareas/mappings

- Descarga de estado de la memoria de la JVM

http://localhost:8080/ListadoDeTareas/heapdump

- Beans de la aplicacion

http://localhost:8080/ListadoDeTareas/beans

Se pueden configurar las funcionalidades para que sean privadas, modificando la propiedad **sensitive** del endpoint

```
endpoints:
  info:
    sensitive: true
```

Si son privadas, se necesitará configurar **Spring Security** para definir el origen de la autenticacion.

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

Una vez añadido, se han de configurar las siguientes propiedades

```
security.user.name=admin

security.user.password=secret

security.user.role=SUPERUSER

management.security.role=SUPERUSER
```

Para desactivar la seguridad

```
management.security.enabled=false
```

Endpoint Custom

Se pueden añadir nuevos EndPoints a la aplicación para que muestren algún tipo de información, para ello basta definir un Bean de Spring que extienda la clase **AbstractEndPoint**

```
@Component
public class ListEndPoints extends AbstractEndpoint<List<Endpoint>> {

    private List<Endpoint> endpoints;

    @Autowired
    public ListEndPoints(List<Endpoint> endpoints) {
        super("listEndpoints");
        this.endpoints = endpoints;
    }

    public List<Endpoint> invoke() {
        return this.endpoints;
    }
}
```

TIP	Solo esta implementacion, puede dar error, por encontrar valores en los Bean a Null, y el parser de Jackson no aceptarlo, para solventarlo, se puede definir en el application.properties la propiedad **spring.jackson.serialization.FAIL_ON_EMPTY_BEANS** a **false**

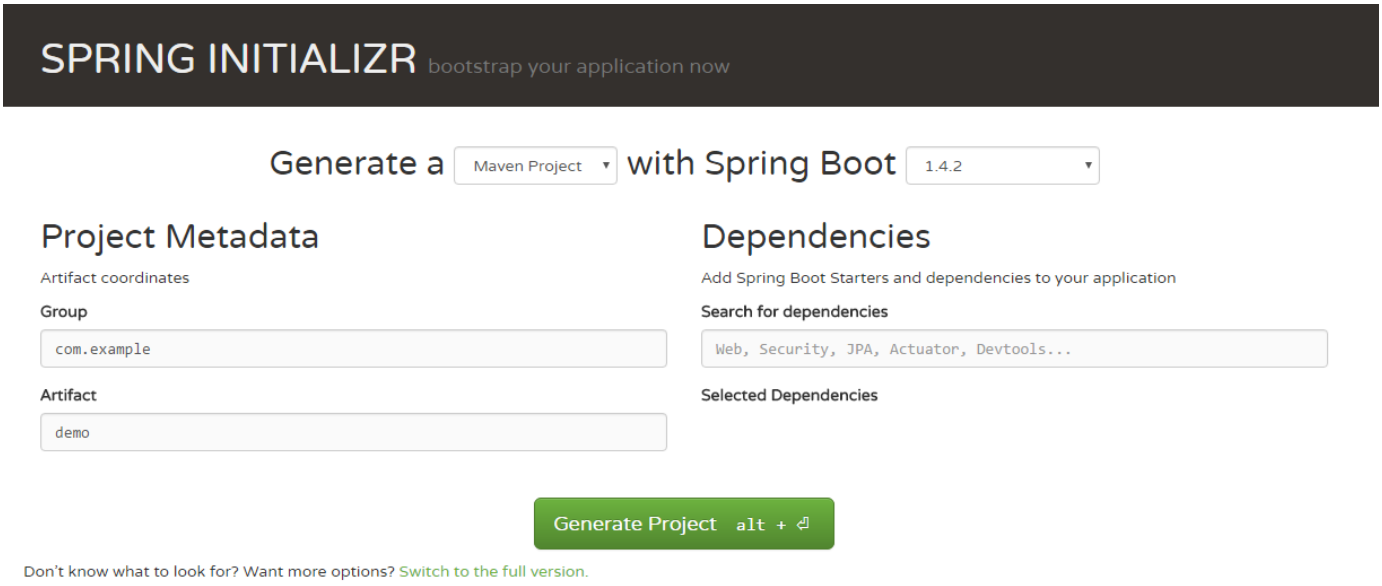
Uso de Java con start.spring.io

Es uno de los modos de emplear el API de **Spring Initializr**, al que tambien se tiene acceso desde

- Spring Tool Suite
- IntelliJ IDEA
- Spring Boot CLI

Es una herramienta que permite crear estructuras de proyectos de forma rapida, a través de plantillas.

Desde la pagina start.spring.io se puede generar una plantilla de proyecto.



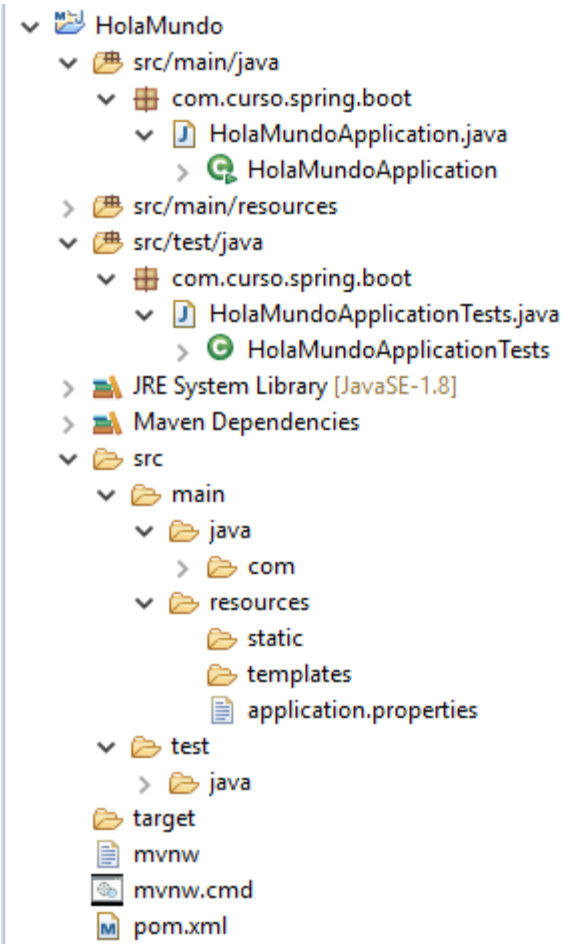
Lo que se ha de proporcionar es

- Tipo de proyecto (Maven o Gradle)
- Versión de Spring Boot
- GroupId
- ArtifactId
- Dependencias

Existe una vista avanzada donde se pueden indicar otros parametros como

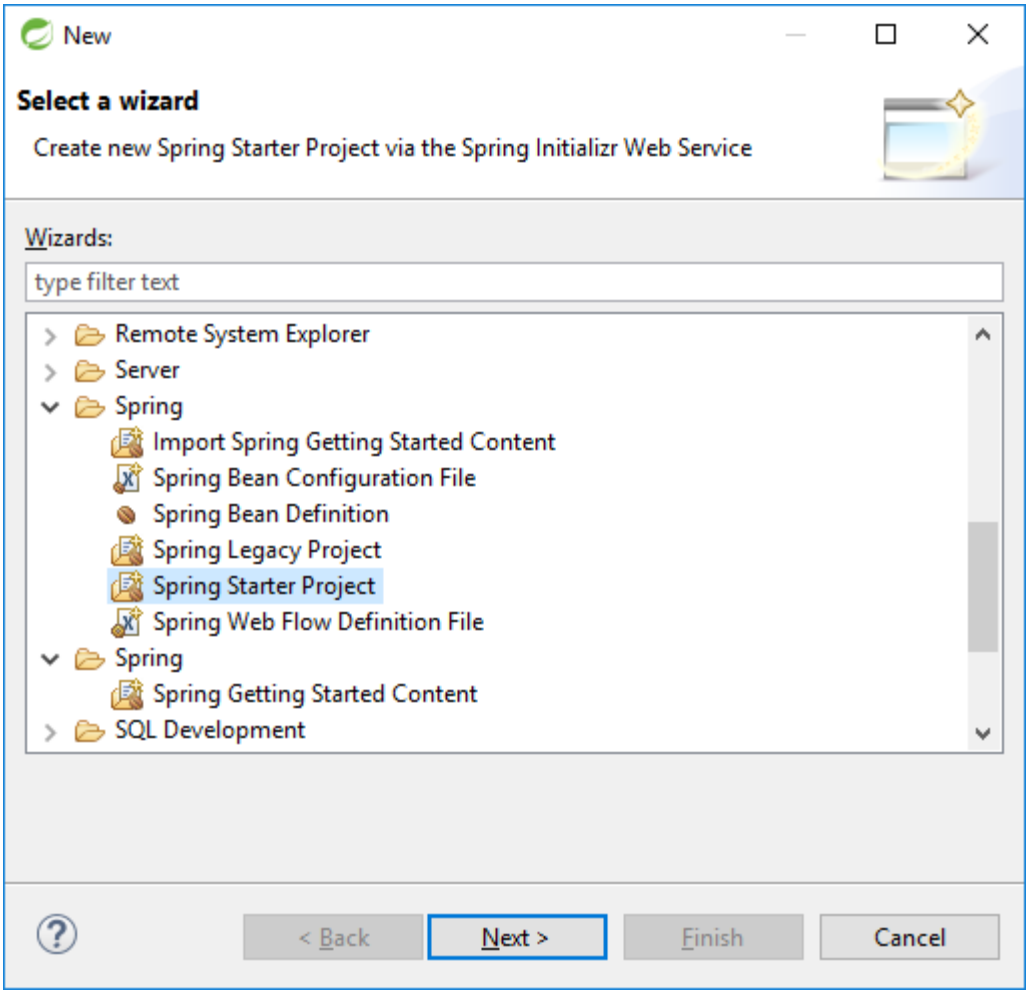
- Versión de java
- El tipo de packaging
- El lenguaje del proyecto
- Seleccion mas detallada de las dependencias

La estructura del proyecto con dependencia web generado será

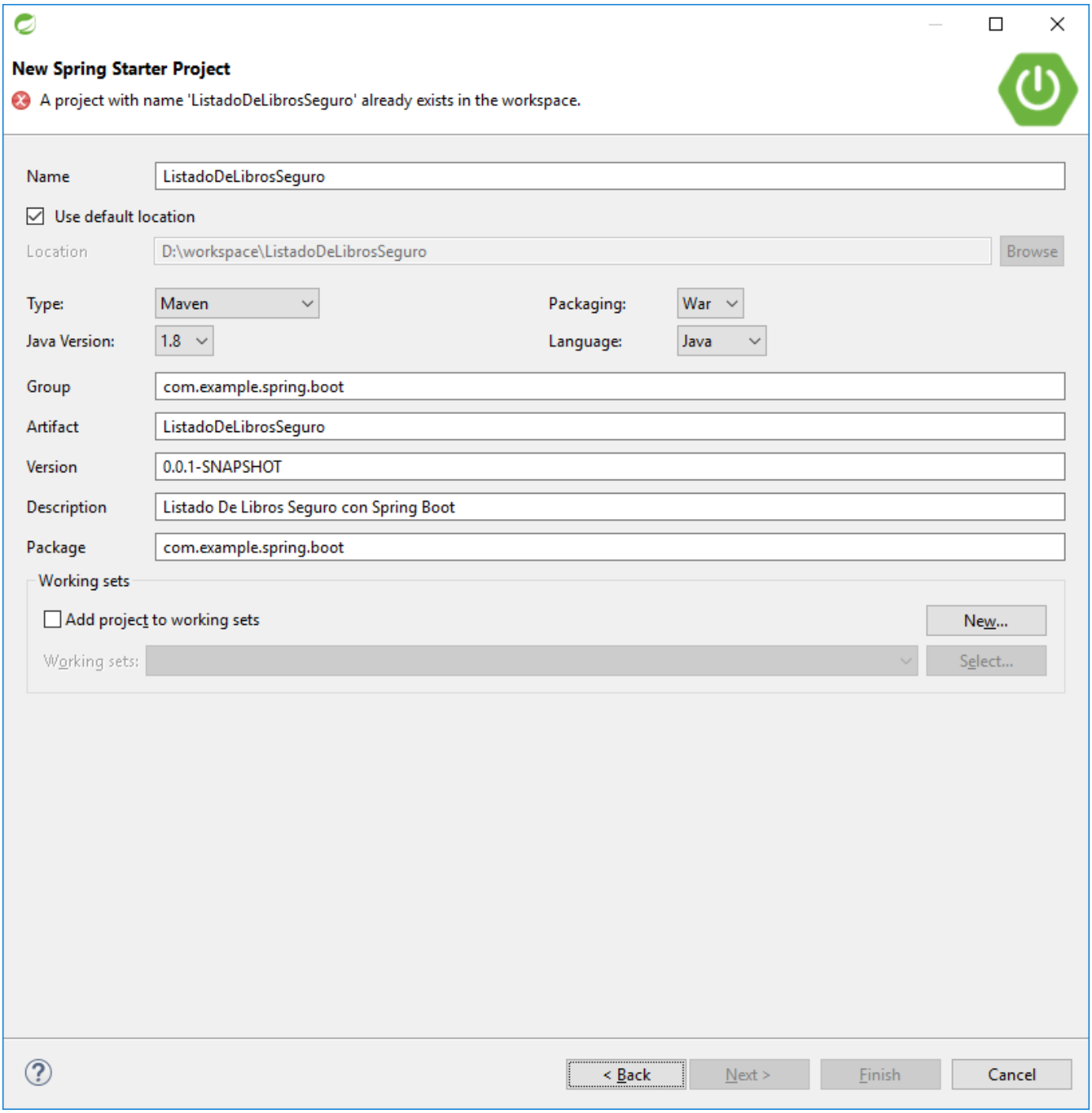


En esta estructura, cabe destacar el directorio **static**, destinado a contener cualquier recurso estatico de una aplicación web.

Desde Spring Tools Suite, se puede acceder a esta misma funcionalidad desde **New** > **Other** > **Spring** > **Spring Starter Project**, es necesario tener internet, ya que STS se conecta a **start.spring.io**



Una vez seleccionada la opcion, se muestra un ofrmulario similar al de la web



Y desde Spring CLI con el comando **init** tambien, un ejemplo de comando seria

```
Spring-CLI# init --build maven --groupId com.ejemplo.spring.boot.web --version 1.0 --java
-version 1.8 --dependencies web --name HolaMundo HolaMundo
```

Que genera la estructura anterior dentro de la carpeta **HolaMundo**

Se puede obtener ayuda sobre los parametros con el comando

```
Spring-CLI# init --list
```

Starters

Son dependencias ya preparadas por Spring, para dotar del conjunto de librerias necesarias para obtener un funcionalidad sin que existan conflictos entre las versiones de las distintas librerias.

Se pueden conocer las dependencias reales con las siguientes tareas

- Maven

```
mvn dependency:tree
```

- Gradle

```
gradle dependencies
```

De necesitarse, se pueden sobrescribir las versiones o incluso excluir librerias, de las que nos proporcionan los **Starter**

- Maven

```
<dependency>
    <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-web</artifactId>
        <exclusions>
            <exclusion>
                <groupId>com.fasterxml.jackson.core</groupId>
            </exclusion>
    </exclusions>
</dependency>
```

- Gradle

```
compile("org.springframework.boot:spring-boot-starter-web") {
    exclude group: 'com.fasterxml.jackson.core'
}
```

Soporte a propiedades

Spring Boot permite configurar unas 300 propiedades, aqui una lista de ellas.

Se pueden configurar los proyectos de Spring Boot unicamnete modificando propiedades, estas se pueden definir en

- Argumentos de la linea de comandos

```
java -jar app-0.0.1-SNAPSHOT.jar --spring.main.show-banner=false
```

- JNDI

```
java:comp/env/spring.main.show-banner=false
```

- Propiedades del Sistema Java

```
java -jar app-0.0.1-SNAPSHOT.jar -Dspring.main.show-banner=false
```

- Variables de entorno del SO

```
SET SPRING_MAIN_SHOW_BANNER=false;
```

- Un fichero **application.properties**

```
spring.main.show-banner=false
```

- Un fichero **application.yml**

```
spring:
    main:
        show-banner: false
```

Las listas en formato **YAML** tiene la siguiente sintaxis

```yaml
security:
  user:
    role:
      - SUPERUSER
      - USER
```

De existir varias de las siguientes, el orden de preferencia es el del listado, por lo que la mas prioritaria es la linea de comandos.

Los ficheros **application.properties** y **application.yml** pueden situarse en varios lugares

- En un directorio **config** hijo del directorio desde donde se ejecuta la aplicación.
- En el directorio desde donde se ejecuta la aplicación.
- En un paquete **config** del proyecto
- En la raiz del classpath.

Siendo el orden de preferencia el del listado, si aparecieran los dos ficheros, el **.propertes** y el **.yml**, tiene prioridad el properties.

Algunas de las propiedades que se pueden definir son:

- **spring.main.show-banner** → Mostrar el banner de spring en el log (por defecto true).
- **spring.thymeleaf.cache** → Deshabilitar la cache del generador de plantillas thymeleaf
- **spring.freemarker.cache** → Deshabilitar la cache del generador de plantillas freemarker
- **spring.groovy.template.cache** → Deshabilitar la cache de plantillas generadas con groovy
- **spring.velocity.cache** → Deshabilitar la cache del generador de plantillas velocity
- **spring.profiles.active** → Perfil activado en la ejecución

TIP	La cache de las plantillas, se emplea en producción para mejorar el rendimiento, pero se debe desactivar en desarrollo ya que sino se ha de parar el servidor cada vez que se haga un cambio en las plantillas.

Configuracion del Servidor

- **server.port** → Puerto del Contenedor Web donde se exponen los recursos (por defecto 8080, para ssl 8443).
- **server.contextPath** → Permite definir el primer nivel del path de las url para el acceso a la aplicacion (Ej: /resource).
- **server.ssl.key-store** → Ubicación del fichero de certificado (Ej: file:///path/to/mykeys.jks).
- **server.ssl.key-store-password** → Contraseña del almacen.

- **server.ssl.key-password** → Contraseña del certificado.

TIP

Para generar un certificado, se puede emplear la herramienta keytool* que incluye la jdk

```
keytool -keystore mykeys.jks -genkey -alias tomcat -keyalg RSA
```

Configuracion del Logger

- **logging.level.root**→ Nivel del log para el log principal (Ej: WARN)
- **logging.level.<paquete>**→ Nivel del log para un log particular (Ej: logging.level.org.springframework.security: DEBUG)
- **logging.path** → Ubicacion del fichero de log (Ej: /var/logs/)
- **logging.file** → Nombre del fichero de log (Ej: miApp.log)

Configuracion del Datasource

- **spring.datasource.url** → Cadena de conexión con el origen de datos por defecto de la auto-configuración (Ej: jdbc:mysql://localhost/test)
- **spring.datasource.username** → Nombre de usuario para conectar al origen de datos por defecto de la auto-configuración (Ej: dbuser)
- **spring.datasource.password** → Password del usuario que se conecta al origen de datos por defecto de la auto-configuración (Ej: dbpass)
- **spring.datasource.driver-class-name**→ Driver a emplear para conectra con el origen de datos por defecto de la auto-configuración (Ej: com.mysql.jdbc.Driver)
- **spring.datasource.jndi-name** → Nombre JNDI del datasorce que se quiere emplear como origen de datos por defecto de la auto-configuración.
- **spring.datasource.name**→ El nombre del origen de datos
- **spring.datasource.initialize**→ Whether or not to populate using data.sql (default:true)
- **spring.datasource.schema** → The name of a schema (DDL) script resource
- **spring.datasource.data**→ The name of a data (DML) script resource
- **spring.datasource.sql-script-encoding**→ The character set for reading SQL scripts
- **spring.datasource.platform**→ The platform to use when reading the schema resource (for example, "schema-{platform}.sql")
- **spring.datasource.continue-on-error**→ Whether or not to continue if initialization fails (default: false)
- **spring.datasource.separator**→ The separator in the SQL scripts (default: ;)
- **spring.datasource.max-active**→ Maximum active connections (default: 100)

- **spring.datasource.max-idle** → Maximum idle connections (default: 8)
- **spring.datasource.min-idle** → Minimum idle connections (default: 8)
- **spring.datasource.initial-size** → The initial size of the connection pool (default: 10)
- **spring.datasource.validation-query** → A query to execute to verify the connection
- **spring.datasource.test-on-borrow** → Whether or not to test a connection as it's borrowed from the pool (default: false)
- **spring.datasource.test-on-return** → Whether or not to test a connection as it's returned to the pool (default: false)
- **spring.datasource.test-while-idle** → Whether or not to test a connection while it is idle (default: false)
- **spring.datasource.max-wait** → The maximum time (in milliseconds) that the pool will wait when no connections are available before failing (default: 30000)
- **spring.datasource.jmx-enabled** → Whether or not the data source is managed by JMX (default: false)

TIP	Solo se puede configurar un unico datasource por auto-configuración, para definir otro, se ha de definir el bean correspondiente

Custom Properties

Se puede definir nuevas propiedades y emplearlas en la aplicación dentro de los Bean.

- Para ello se ha de definir, dentro de un Bean de Spring, un atributo de clase que refleje la propiedad y su método de SET

```
private String prefijo;
public void setPrefijo(String prefijo) {
    this.prefijo = prefijo;
}
```

- Para las propiedades con nombre compuesto, se ha de configurar el prefijo con la anotación **@ConfigurationProperties** a nivel de clase

```
@Controller
@RequestMapping("/")
@ConfigurationProperties(prefix="saludo")
public class HolaMundoController {}
```

- Ya solo falta definir el valor de la propiedad en **application.properties** o en **application.yml**

```yaml
saludo:
  prefijo: Hola
```

TIP	Para que la funcionalidad de properties funcione, se debe añadir **@EnableConfigurationProperties**, pero con Spring Boot no es necesario, ya que está incluido por defecto.

Otra opcion para emplear propiedades, es el uso de la anotacion **@Value** en cualquier propiedad de un bean de spring, que permite leer la propiedad si esta existe o asignar un valor por defecto en caso que no exista.

```java
@Value("${message:Hello default}")
private String message;
```

Profiles

Se pueden anotar **@Bean** con **@Profile**, para que dicho Bean sea solo añadido al contexto de Spring cuando el profile indicado esté activo.

```java
@Bean
@Profile("production")
public DataSource dataSource() {
    DataSource ds = new DataSource();
    ds.setDriverClassName("org.mysql.Driver");
    ds.setUrl("jdbc:mysql://localhost:5432/test");
    ds.setUsername("admin");
    ds.setPassword("admin");
    return ds;
}
```

Tambien se puede definir un conjunto de propiedades que solo se empleen si un perfil esta acivo, para ello, se ha de crear un nuevo fichero **application-{profile}.properties**.

En el caso de los ficheros de YAML, solo se define un fichero, el **application.yml**, y en el se definen todos los perfiles, separados por **---**

```yaml
---
spring:
    profiles: production
    datasource:
        url: jdbc:mysql://localhost:5432/test
        username: admin
        password: admin
        jpa:
            database-platform: org.hibernate.dialect.MySQLDialect
```

Para activar un **Profile**, se emplea la propiedad **spring.profiles.active**, la cual puede establecerse como:

- Variable de entorno

```
SET SPRING_PROFILES_ACTIVE=production;
```

- Con un parametro de inicio

```
java -jar applicacion-0.0.1-SNAPSHOT.jar --spring.profiles.active=production
```

TIP De definirse mas de un perfil activo, se indicaran con un listado separado por comas

JPA

Al añadir el starter de JPA, por defecto Spring Boot va a localizar todos los Bean detro del paquete y subpaquetes donde se encuentra la clase anotada con **@SpringBootApplication** en busca de interfaces Repositorio, que extiendan la interface **JpaRepository**, de no encontrarse la interface que define el repositorio dentro del paquete o subapaquetes, se puede referenciar con **@EnableJpaRepositories**

Cuando se emplea JPA con Hibernate como implementación, esté último tiene la posibilidad de configurar su comportamiento con respecto al schema de base de datos, pudiendo indicarle que lo cree, que lo actualice, que lo borre, que lo valide... esto se consigue con la propiedad **hibernate.ddl-auto**

```yaml
spring:
    jpa:
        hibernate:
            ddl-auto: validate
```

NOTE

Los posibles valores para esta propiedad son:

- none → This is the default for MySQL, no change to the database structure.
- update → Hibernate changes the database according to the given Entity structures.
- create → Creates the database every time, but don't drop it when close.
- create-drop → Por defecto para H2. the database then drops it when the SessionFactory closes.

Habrá que añadir al classpath, con dependencias de Maven, el driver de la base de datos a emplear, Spring Boot detectará el driver añadido y conectará con una base de datos por defecto.

La dependencia para MySQL será

```
 <dependency>
    <groupId>mysql</groupId>
    <artifactId>mysql-connector-java</artifactId>
</dependency>
```

NOTE

Las versiones de algunas depedencias no es necesario que se indiquen en Spring Boot, ya que vienen predefinidas en el **parent**

Para configurar un nuevo origen de datos, se indican las siguientes propiedades

```
spring.jpa.hibernate.ddl-auto=create
spring.datasource.url=jdbc:mysql://localhost:3306/db_example
spring.datasource.username=springuser
spring.datasource.password=ThePassword
```

Errores

Por defecto Spring Boot proporciona una pagina para represenar los errores que se producen en las aplicaciones llamada **whitelabel**, para sustituirla por una personalizada, basta con definir alguno de los siguientes componentes

- Cuanlquier Bena que implemente **View** con Id **error**, que será resuelto por **BeanNameViewResolver**.
- Plantilla **Thymeleaf** llamada **error.html** si **Thymeleaf** esta configurado.
- Plantilla **FreeMarker** llamada **error.ftl** si **FreeMarker** esta configurado.
- Plantilla **Velocity** llamada **error.vm** si **Velocity** esta configurado.
- Plantilla **JSP** llamada **error.jsp** si se emplean vistas JSP.

Dentro de la vista, se puede acceder a la siguiente información relativa al error

- **timestamp** → La hora a la que ha ocurrido el error
- **status** → El código HTTP
- **error** → La causa del error
- **exception** → El nombre de la clase de la excepción.
- **message** → El mensaje del error
- **errors** → Los errores si hay mas de uno
- **trace** → La traza del error
- **path** → La URL a la que se accedia cuando se produjo el error.

Seguridad de las aplicaciones

Para añadir Spring security a un proyecto, habrá que añadir

- En Maven

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

- En Gradle

```
compile("org.springframework.boot:spring-boot-starter-security")
```

Al añadir Spring Security al Classpath, automaticamente Spring Boot, hace que la aplicación sea segura, nada es accesible.

Se creará un usuario por defecto **user** cuyo password e generará cada vez que se arranque la aplicación y se pintará en el log

```
Using default security password: ce9dadfa-4397-4a69-9fc7-af87e0580a10
```

Evidentemente esto es configurable, dado que cada aplicación, tendrá sus condiciones de seguridad, para establecer la configuración se puede añadir una nueva clase de configuración, anotada con **@Configuration** y además para que permita configurar la seguridad, debe estar anotada con **@EnableWebSecurity** y extender de **WebSecurityConfigurerAdapter**

```
@Configuration
@EnableWebSecurity
public class SecurityConfig extends WebSecurityConfigurerAdapter {
    @Autowired
    private ReaderRepository readerRepository;

    @Override
    protected void configure(HttpSecurity http) throws Exception {
        http
            .authorizeRequests()
            .antMatchers("/").access("hasRole('READER')")
            .antMatchers("/**").permitAll()
            .and()
            .formLogin()
            .loginPage("/login")
            .failureUrl("/login?error=true");
    }
    @Override
    protected void configure(AuthenticationManagerBuilder auth) throws Exception {
        auth
            .userDetailsService(new UserDetailsService() {
                @Override
                public UserDetails loadUserByUsername(String username) throws
UsernameNotFoundException {
                    return readerRepository.findOne(username);
                }
            });
    }
}
```

En esta clase, se puede configurar tanto los requisitos para acceder a recursos via web (autorización), como la vía de obtener los usuarios validos de la aplicación (autenticación), como otras configuraciones propias de la seguridad, como SSL, la pagina de login, ...

Soporte Mensajeria JMS

Para emplear JMS de nuevo Spring Boot, proporciona un starter, en este caso para varias tecnologias: ActiveMQ, Artemis y HornetQ

Para añadir por ejemplo ActiveMQ, se añadirá a dependencia Maven

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

Una vez Spring Boot encuentra en el classpath el jar de **ActiveMQ**, creará los objetos necesarios para conectarse al recurso JMS **Topic/Queue**, simplemente hará falta configurar las siguientes propiedades para inicar donde se encuentra el endpoint de **ActiveMQ**

- **spring.activemq.broker-url** → (Ej: tcp:/localhost:61616)
- **spring.activemq.user** → (EJ: admin)
- **spring.activemq.password** → (Ej: admin)

NOTE

Se espera que este configurado un endpoint de ActiveMQ, se puede descargar la distribución de aquí.

Para arrancarlo se ha de ejecutar el comando **/bin/activemq start** que levanta el servicio en local con los puertos **8161** para la consola administrativa y **61616** para la comunicacion de con los clientes.

El usuario y password por defecto son **admin/admin**

Como es habitual en las aplicaciones **Boot**, no será necesario añadir la anotacion **@EnableJms** a la clase de aplicación, ya que estará contemplada con **@SpringBootApplication**.

Consumidores

Para definir un Bean que consuma los mensajes del servicio JMS, se emplea la anotación **@JmsListener**

```
@Component
public class Receiver {
    @JmsListener(destination = "mailbox")
    public void receiveMessage(Email email) {
        System.out.println("Received <" + email + ">");
    }
}
```

NOTE Debera existir un **Queue** o **Topic** denominado **mailbox**

Productores

Para definir un Bean que envie mensajes se empleará un **Bean** creado por Spring de tipo **JmsTemplate**, empleando las funcionalidades **send** o **convertAndSend**.

```
@Component
public class MyBean {
    private JmsTemplate jmsTemplate;
    @Autowired
    public MyBean(JmsTemplate jmsTemplate) {
        this.jmsTemplate = jmsTemplate;
    }
}
```

Se puede redefinir la factoria de contenedores

```
@Bean
public JmsListenerContainerFactory<?> myFactory(ConnectionFactory connectionFactory,

DefaultJmsListenerContainerFactoryConfigurer configurer) {
    DefaultJmsListenerContainerFactory factory = new DefaultJmsListenerContainerFactory(
);
    // This provides all boot's default to this factory, including the message converter
    configurer.configure(factory, connectionFactory);
    // You could still override some of Boot's default if necessary.
    return factory;
}
```

Indicandolo posteriormente en los listener

```
@Component
public class Receiver {
    @JmsListener(destination = "mailbox", containerFactory = "myFactory")
    public void receiveMessage(Email email) {
        System.out.println("Received <" + email + ">");
    }
}
```

Soporte Mensajeria AMQP

AMQP es una especificacion de mensajeria asincrona, donde todos los bytes transmitidos son especificados, por lo que se pueden crear implementaciones en distintas plataformas y lenguajes.

La principal diferencia con JMS, es que mientras que en JMS los productores pueden publicar mensajes sobre: * **Queue** (un consumidor) * y **Topics** (n consumidores)

En AMQP solo hay **Queue** (un receptor), pero se incluye una capa por encima de estas **Queue**, los **Exchange**, que es donde se publican los mensajes por parte de los productores y estos **Exchange**, tienen la capacidad de publicar los mensajes que les llegan en una sola **Queue** o en varias, emulando los dos comportamientos de JMS.

Para trabajar con AMQP, se necesita un servidor de AMQP, como **RabbitMQ**, para instalarlo se necesita instalar Erlang a parte de RabbitMQ

Además de instalar **Erlang**, habra que definir a variable de entorno **ERLANG_HOME**.

La configuracion por defecto de **RabbitMQ** es escuchar por el puerto 5672

Receiver

Una vez instalado el Bus, se necesitará un proyecto que defina los **Receiver**, para ello se ha de incluir la dependencia

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

Y añadir un **Bean** al contexto de Spring que haga de **Receiver**, no tiene porque implementar ningun API.

```
@Component
public class Receiver {
    public void receiveMessage(String message) {
        System.out.println("Received <" + message + ">");
    }
}
```

Este **Bean**, se ha de registrar como **Receiver** AMQP, para ello lo primero es indicar que método se ha de ejecutar, para ello se emplea la clase **MessageListenerAdapter**.

```
@Bean
public MessageListenerAdapter listenerAdapter(Receiver receiver) {
    return new MessageListenerAdapter(receiver, "receiveMessage");
}
```

En segundo lugar, hay que definir los objetos que definen la estructura del Broker

- Queue
- Exchange
- Binding

```
final static String queueName = "spring-boot";

@Bean
public Queue queue() {
    return new Queue(queueName, false);
}

@Bean
public TopicExchange exchange() {
    return new TopicExchange("spring-boot-exchange");
}

@Bean
public Binding binding(Queue queue, TopicExchange exchange) {
    return BindingBuilder.bind(queue).to(exchange).with(queueName);
}
```

Y por último asociar el **Receiver**, con la estructura del **Broker** a traves de un **ConnectionFactory**, que creará Spring gracias a los Beans anteriormente configurados

```
@Bean
public SimpleMessageListenerContainer container(ConnectionFactory connectionFactory,
MessageListenerAdapter listenerAdapter) {
    SimpleMessageListenerContainer container = new SimpleMessageListenerContainer();
    container.setConnectionFactory(connectionFactory);
    container.setQueueNames(queueName);
    container.setMessageListener(listenerAdapter);
    return container;
}
```

Producer

Para producir nuevos mensajes, se emplea la clase **RabbitTemplate**, que permite enviar mensajes con métodos **convertAndSend**

```
rabbitTemplate.convertAndSend(ConfiguracionAMQP.queueName, "Hello from RabbitMQ!");
```

Este objeto, será creado por el contexto de Spring con los Bean que

Testing

Para realizar pruebas en las aplicaciones Spring Boot, se ha de añadir el starter de test

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

Y crear clases de Test, anotadas con

```
@RunWith(SpringRunner.class)
@SpringBootTest
public class SmokeTest {
}
```

La anotacion **@SpringBootTest**, crea el contexto de Spring empleando las clase aplicación de Spring Boot, aquella anotada con **@SpringBootApplication**, por lo que el contexto para las pruebas estará compuesto por los mismos beans que el de la aplicación.

Dado que el test tiene el mismo contexto que la aplicación, se puede obtener cualquier bean definido en el contexto de Spring para realizar el test,con la anotacion **@Autowired**

```
@RunWith(SpringRunner.class)
@SpringBootTest
public class SmokeTest {
    @Autowired
    private HomeController controller;
}
```

NOTE	Una caracteritisca de estos Test, es que cachean el contexto de Spring a lo largo de la ejecucion de todos los metodos de testing, aunque esto se puede controlar con **@DirtiesContext**

NOTE

El Starter de Test, incluye la libreria **AssertJ**, que permite definir predicados para las aserciones, basandose en los métodos estaticos de la clase **org.assertj.core.api.Assertions**. Estos predicados son más legibles que los que se pueden definir con la libreria **JUnit** y los métodos estaticos de la clase **org.junit.Assert**

```
assertThat(this.restTemplate.getForObject("http://localhost:8080/", String
.class)).contains("Hello World");

assertThat(controller).isNotNull();
```

Testing Web

Para testear la capa Web de una aplicacion Spring Boot, se ha de configurar la propiedad **webEnvironment** de la anotacion **@SpringBootTest**, pudiendo tener los siguientes valores

- **WebEnvironment.DEFINED_PORT** → Para emplear la configuracion definida en el contexto de Spring
- **WebEnvironment.NONE** →
- **WebEnvironment.MOCK** →
- **WebEnvironment.RANDOM_PORT** → Genera un puerto de escucha de forma aleatoria, pudiendo obtener el puerto generado por inyeccion dentro de los Test

```
@RunWith(SpringRunner.class)
@SpringBootTest(webEnvironment = WebEnvironment.RANDOM_PORT)
public class HttpRequestTest {

    @LocalServerPort
    private int port;
}
```

Una vez configurada la propiedad **webEnvironment**, en el contexto de Spring se dispone de un Bean de tipo **TestRestTemplate**, que funciona de forma similar a **RestTemplate**.

```
@Autowired
private TestRestTemplate restTemplate;
```

Las pruebas anteriores, serían pruebas de humo o de integración total.

Se pueden definir Mocks de Beans ya existentes en el contexto, empleando dichos Mock en lugar de los Beans del contexto

```
@MockBean
private GreetingService service;

@Before
public void setup() {
    when(service.greet()).thenReturn("Hello Mock");
}
```

Esto puede tener sentido en momentos puntuales, dado que se siguen creando los Beans del contexto, no siendo empleados alguno de ellos, por lo que no es muy recomendable.

Pruebas de Integración

Se pueden realizar pruebas de integración parcial Mockeando el servidor, es decir sin levantar el servidor, para ello se ha de emplear la anotacion **@AutoConfigureMockMvc**, que define un nuevo Bean de Spring de tipo **MockMvc**, que permite realizar los accesos a la capa web.

```
@AutoConfigureMockMvc
public class ApplicationTest {

    @Autowired
    private MockMvc mockMvc;
}
```

El objeto **mockMvc** permite realizar acciones definiendolas con el método **perform()**, siendo las posibles acciones las definidas en la clase **org.springframework.test.web.servlet.request.MockMvcRequestBuilders**, que equivalen a los **METHOD HTTP**.

```
this.mockMvc.perform(get("/"));
```

Una vez realizada la acción, se puede acceder a los resultados de la acción para validarlos, para ello se emplean los **org.springframework.test.web.servlet.result.MockMvcResultHandlers**, que dan acceso al status, el contenido, las cookies, las cabeceras, ... de la respuesta de la teorica petición realizada.

```
this.mockMvc.perform(get("/"))
                .andDo(print())
                .andExpect(status().isOk())
                .andExpect(content().string(containsString("Hello World")));
```

Pruebas Unitarias

Tambien se pueden realizar pruebas unitarias sobre la capa de controladores, de nuevo mockeando el servidor, y ademas mockeando la capa de servicios de la que dependen los controladores, para ello se evita arrancar todo el contexto de spring, centrandose en crear solo los Bean a inspecciona, los controladores y **Mocks** de la capa de Servicio, para ello se ha de definir la anotación **@WebMvcTest** en lugar de la tupla **@SpringBootTest** y **@AutoConfigureMockMvc**

```
@RunWith(SpringRunner.class)
@WebMvcTest
public class MockMvcTest {
}
```

La anotacion **@WebMvcTest** permite indicar que controladores se quieren instanciar para las pruebas, de no indicarse, se instanciarán todos los que se encuentren dentro del paquete base de la aplicación.

La capa de servicio se puede mockear empleando la anotacion **@MockBean**, que crea un objeto Mock de **Mockito**, el cual habrá que configurar.

```
@RunWith(SpringRunner.class)
@WebMvcTest
public class MockMvcTest {
    @MockBean
    private GreetingService service;

    @Before
    public void setup() {
        when(service.greet()).thenReturn("Hello Mock");
    }
}
```

Como en el caso anterior, se define un bean de tipo **MockMvc**.

Pruebas Integracion parcial: Capa de Persistenca con BD

Para la configuracion del contexto de ejecución de los repositorios de **JPA Data**, se tiene la anotacion **@DataJpaTest**

```
@RunWith(SpringRunner.class)
@DataJpaTest
public class ExampleRepositoryTests {

    @Autowired
    private TestEntityManager entityManager;

    @Autowired
    private UserRepository repository;

    @Test
    public void testExample() throws Exception {
        this.entityManager.persist(new User("sboot", "1234"));
        User user = this.repository.findByUsername("sboot");
        assertThat(user.getUsername()).isEqualTo("sboot");
        assertThat(user.getVin()).isEqualTo("1234");
    }

}
```

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 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"/>
```

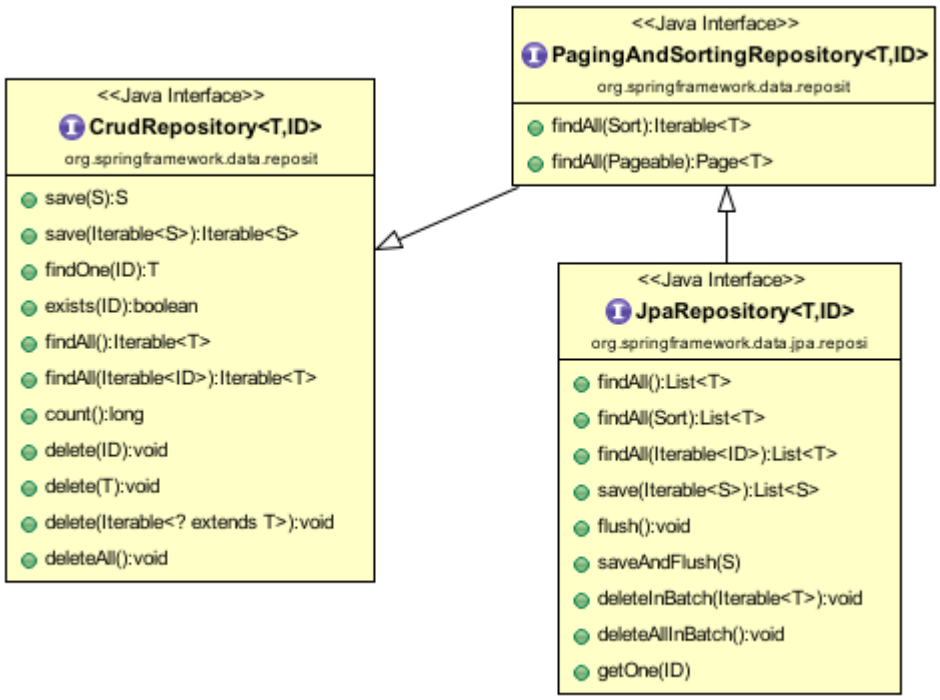
Este bean, se encarga de capturar todas las excepciones lanzadas por clases anotadas con @repository y realizar la conversión.

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

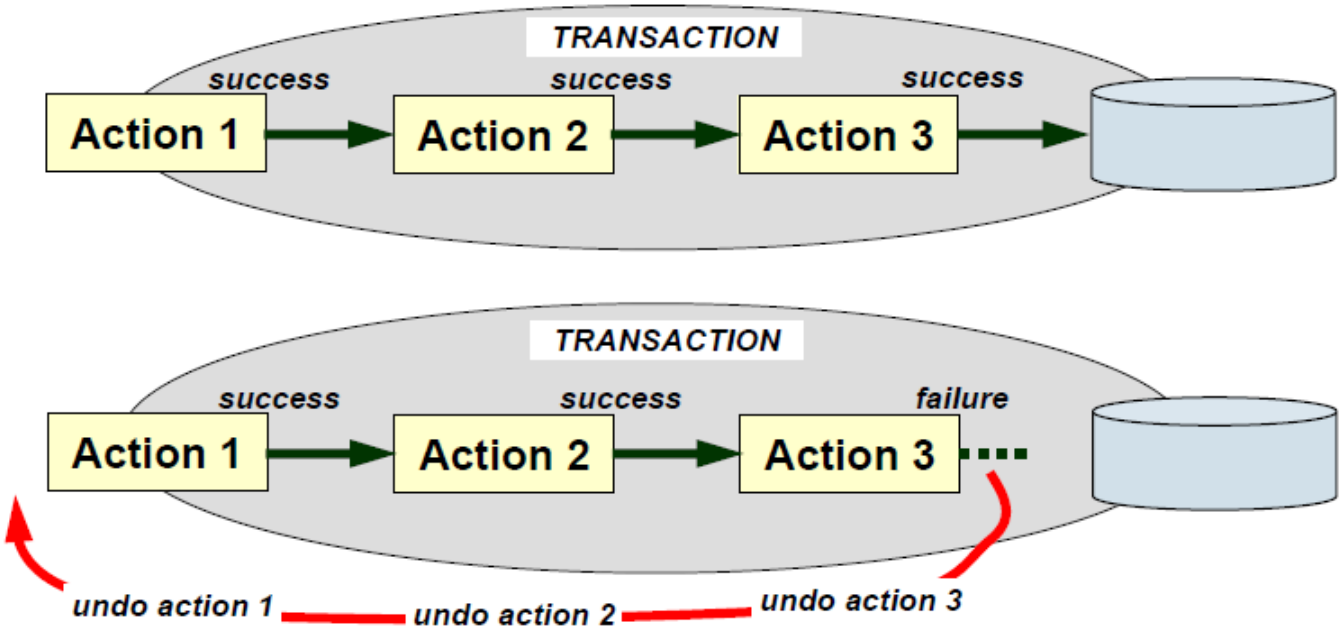
```
<dependency>
    <groupId>com.querydsl</groupId>
    <artifactId>querydsl-apt</artifactId>
</dependency>

<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

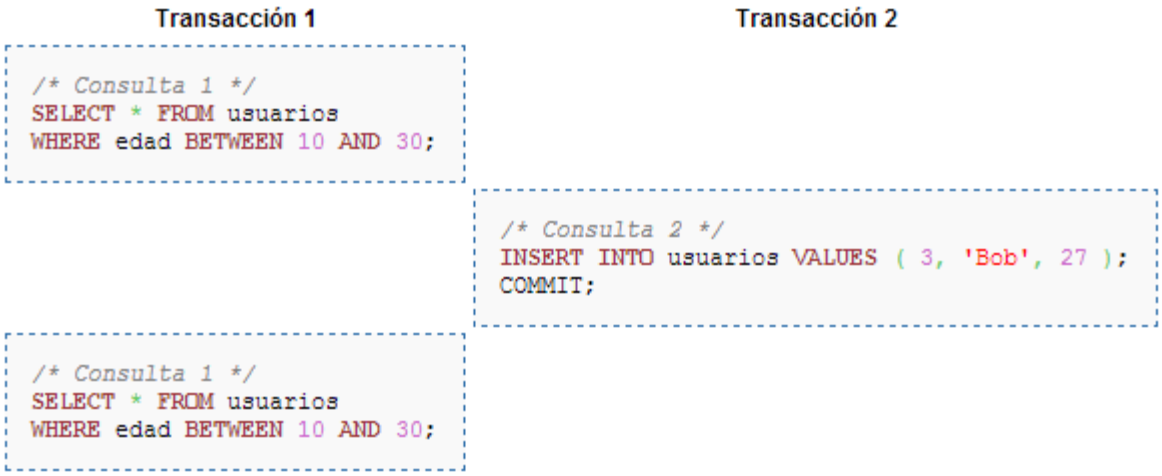
```sql
/* Consulta 1 */
SELECT * FROM usuarios WHERE id = 1;
```

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 Cache

Funcionalidad de Spring que permite cachear las respuestas de métodos.

Especialmente interesante antes repositorios que no alteren su contenido a lo largo del tiempo y que se consulten de forma recurrente en la aplicación.

Esta incluida en el modulo de context.

```
<dependency>
    <groupId>org.springframework</groupId>
    <artifactId>spring-context</artifactId>
    <version>5.0.6.RELEASE</version>
</dependency>
```

Se basa en la definición de un Bean de Spring de tipo **org.springframework.cache.CacheManager** del cual se dispone las siguientes implementaciones

- SimpleCacheManager
- CompositeCacheManager
- ConcurrentMapCacheManager
- NoOpCacheManager

La mas completa seria **ConcurrentMapCacheManager**.

```
@Bean
public CacheManager cacheManager(){
    return new ConcurrentMapCacheManager();
}
```

Este Bean se encarga de gestionar el almacen de caches y todas las operaciones relativas a ellas.

Para crear un nuevo almacen, se ha de emplear la anotacion **@Cacheable**

```
@Cacheable("fecha")
public Date getFecha() {
    return new Date();
}
```

Se puede invalidar la cache invocando un método que esté anotado con **@CacheEvict**

```
@CacheEvict(value="fecha", allEntries=true)
public void resetear(){
    _log.info("Resetando la cache");
}
```

Se puede actualizar de forma parcial la cache con los resultados retornados por un método

```
@CachePut(value="fecha")
public Date actualizar() {}
```

Para activar estas anotaciones es necesario incluir la anotacion **@EnableCaching** en una clase de configuracion.

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

Con configuracion basada en XML, se haria

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

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

```java
@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

```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

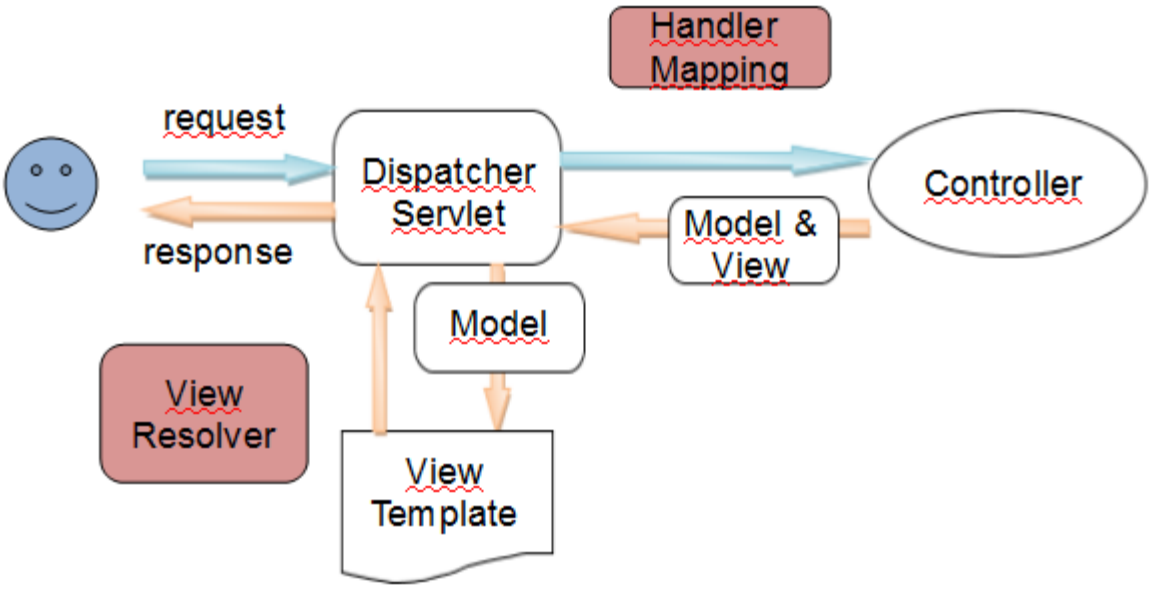
Spring MVC, como su nombre indica es un framework que implementa Modelo-Vista-Controlador, esto quiere decir que proporcionará componentes especializados en cada una de esas tareas.

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");
```

Spring Security

La seguridad en Spring:

- Basada en otorgar el acceso.
- Jerárquica y perimetral. Aplica niveles.
- Transportable.

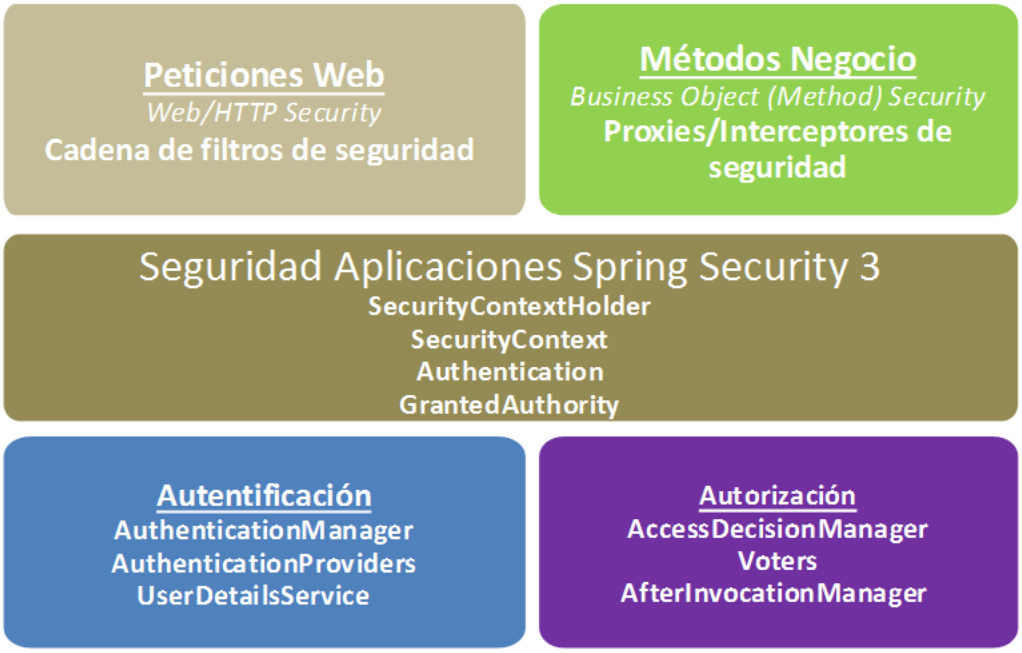
NOTE	Perimetral = estas dentro o no. Jerarquica = se pueden aplicar niveles de acceso a los contenidos.

La seguridad en JEE:

- Basada en restricciones.
- Perimetral.
- Dificil migrar.

NOTE	Restricciones = Todo es accesible hasta que se restringe el acceso. No es estandar dentro de los contenedores JEE, no es facil migrar.

Arquitectura



NOTE

Acceder a la app de fbi.war sin seguridad, se ve que accede hasta el fondo, a todas las funcionalidades sin restriccion.

localhost:8081/fbi → "mostrar expedientes" → "clasificar" → "desclasificar" → "mostrar"

Dependencias con Maven

Se han de añadir las siguientes dependencias al proyecto.

```
<dependency>
    <groupId>org.springframework.security</groupId>
    <artifactId>spring-security-taglibs</artifactId>
    <version>3.2.4.RELEASE</version>
</dependency>
<dependency>
    <groupId>org.springframework.security</groupId>
    <artifactId>spring-security-config</artifactId>
    <version>3.2.4.RELEASE</version>
</dependency>
<dependency>
    <groupId>org.springframework.security</groupId>
    <artifactId>spring-security-crypto</artifactId>
    <version>3.2.4.RELEASE</version>
</dependency>
```

Filtro de seguridad

La seguridad con Spring, se basa en una clasede Spring **FilterChainProxy**, este filtro no será más que un Bean de Spring.

```
<bean id="springSecurityFilterChain" class=
"org.springframework.security.web.FilterChainProxy">
```

NOTE	El Bean de Spring de tipo **FilterChainProxy**, no es necesario definirlo directamente, ya que es una de las configuraciones por defecto que se añaden al activar la seguridad en Spring.

Sobre este Bean, delegará un Filtro Web especial, el **DelegatingFilterProxy**, que como su nombre indica, delega en el contexto de Spring lo que intercepta.

Con XML

```
<filter>
    <display-name>springSecurityFilterChain</display-name>
    <filter-name>springSecurityFilterChain</filter-name>
    <filter-class>
        org.springframework.web.filter.DelegatingFilterProxy
    </filter-class>
</filter>
<filter-mapping>
    <filter-name>springSecurityFilterChain</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("springSecurityFilterChain",
            new DelegatingFilterProxy("springSecurityFilterChain"))
                .addMappingForUrlPatterns(null, false, "/*");
        // ...
    }
}
```

Contexto de Seguridad

Se ha de añadir la anotación **@EnableWebSecurity** a una clase con **@Configuration** para que se genere el objeto **WebSecurityConfigurer**, que tiene la configuración por defecto de Spring Security.

Está configuración puede ser sobreescrita haciendo a su vez extender la clase **@Configuration** de **WebSecurityConfigurerAdapter**.

```
@Configuration
@EnableWebSecurity
public class ConfiguracionSpringSecurity extends WebSecurityConfigurerAdapter {

}
```

AuthenticationManagerBuilder

Permite definir de donde se han de obtener los usuarios y roles que se emplearán en la aplicación, para ello sobrescribir el método correspondiente de la clase **WebSecurityConfigurerAdapter**.

```
@Override
protected void configure(AuthenticationManagerBuilder auth) throws Exception {
    auth.inMemoryAuthentication().withUser("Fernando").password("1234").roles("AGENTE");
    auth.inMemoryAuthentication().withUser("Mulder").password("fox").roles(
"AGENTE_ESPECIAL");
    auth.inMemoryAuthentication().withUser("Scully").password("dana").roles(
"AGENTE_ESPECIAL");
    auth.inMemoryAuthentication().withUser("Skinner").password("walter").roles("DIRECTOR
");
}
```

NOTE	El objeto **AuthenticationManagerBuilder**, tiene un método **jdbcAuthentication()** que permite definir la conexion contra una base de datos para obtener los usuarios y roles.

Proteccion de recursos

Permite configurar las seguridad Web sobre las peticiones http.

Patrón de recursos protegido y rol que puede acceder.

```
@Override
protected void configure(HttpSecurity http) throws Exception {
    http
        .authorizeRequests()
            .antMatchers("/**").access("hasAnyRole('AGENTE_ESPECIAL','DIRECTOR')");
}
```

NOTE En este caso, esta limitado el aceso a cualquier recurso a aquellos usuarios que tienen como ROL = ROLE_AGENTE_ESPECIAL, además de que el proceso de Login, se realiza con formulario.

Para excluir recursos de la protección.

```
http
    .authorizeRequests()
        .antMatchers("/paginas/*").permitAll()
        .antMatchers("/css/*").permitAll()
        .antMatchers("/imagenes/*").permitAll();
```

Login

Para definir un proceso de Login a través de formulario con una pagina de login personalizada.

```
http
    .formLogin()
        .loginPage("/paginas/nuestro-login.jsp")
        .failureUrl("/paginas/nuestro-login.jsp?login_error");
```

NOTE Dado que cuando se intenta acceder a un recurso y este está asegurado, se nos redirigirá a la página de login, para conocer a que recurso se queria realmente acceder, Spring crea una **Session** temporal donde almacena la URL.

Logout

Para activar el logout indicando la url para realizar el logout, definir la pagina a la que se redirecciona una vez realizado el logout y el nombre de las cookies que se han de borrar en el proceso.

```
http
    .logout()
        .logoutUrl("/logout")                                                    2
        .invalidateHttpSession(true)
        .logoutSuccessUrl("/paginas/desconectado.jsp")
        .deleteCookies("JSESSIONID");
```

Por tanto para realizar el logout, basta con invocar la url **/logout**

```
<a href="<c:url value='/logout'/>">desconectar</a>
```

CSRF

El CSRF (Cross-site request forgery) o control de accesos desde sitio externos, permite mediante la adición de una huella aleatoria en las transacciones entre servidor y cliente, controlar que no se acceda al servidor desde otro sitio que no sea la propia aplicación.

Consiste en añadir un campo oculto para guardar el token en todos los formularios que utilicen el método POST:

```
<input type="hidden" name="${_csrf.parameterName}" value="${_csrf.token}"/>
```

Este Token, tambien se puede añadir con la libreria de etiquetas de Spring security:

```
<sec:csrfInput />
```

Este control se puede desactivar.

```
    http
        .csrf().disable();
```

UserDetailService

Permite personalizar la forma en la que se realiza la autenticación.

Spring proporciona implementaciones de referencia como **InMemoryUserDetailsManager** o **JdbcUserDetailsManager**

```
@Override
public void configure(AuthenticationManagerBuilder auth) throws Exception {
    auth.userDetailsService(userDetailsService());
}

public UserDetailsService userDetailsService(){
    Properties usuarios = new Properties();
    usuarios.put("Fernando","1234,ROLE_AGENTE,enabled");
    usuarios.put("Mulder"  ,"fox,ROLE_AGENTE_ESPECIAL,enabled");
    usuarios.put("Scully"  ,"dana,ROLE_AGENTE_ESPECIAL,enabled");
    usuarios.put("Skinner" ,"walter,ROLE_DIRECTOR,enabled");

    return new InMemoryUserDetailsManager(usuarios);
}
```

Encriptación

Se trata de posibilitar el medio para que se almacenen las contraseñas encriptadas, pero que se sigan pudiendo resolver, para ello se ha de definir un **Encoder** y asociarlo al **AuthenticationManagerBuilder**.

```
@Override
public void configure(AuthenticationManagerBuilder auth) throws Exception {
    PasswordEncoder encoder = new BCryptPasswordEncoder();
    auth.userDetailsService(userDetailsService()).passwordEncoder(encoder);
}

public UserDetailsService userDetailsService(){
    Properties usuarios = new Properties();
    usuarios.put("Fernando",
"$2a$10$SMPYtil7Hs2.cV7nrMjrM.dRAkuoLdYM8NdVrF.GeHfs/MrzcQ/zi,ROLE_AGENTE,enabled");
    usuarios.put("Mulder"  ,
"$2a$10$M2JRRHUHTfv4uMR4NWmCLebk1r9DyWSwCMZmuq4LKbImOkfhGFAIa,ROLE_AGENTE_ESPECIAL,enable
d");
    usuarios.put("Scully"  ,
"$2a$10$cbF5xp0grCOGcI6jZvPhA.asgmILATW1hNbM2MEqGJEFnRhhQd3ba,ROLE_AGENTE_ESPECIAL,enable
d");
    usuarios.put("Skinner" ,
"$2a$10$ZFtPIULMcxPe3r/5VunbVujMD7Lw8hkqAmJlxmK5Y1TK3L1bf8ULG,ROLE_DIRECTOR,enabled");

    return new InMemoryUserDetailsManager(usuarios);
}
```

Adicionalmente se puede realizar dicha configuración aprovechando una caracteristicas de las clases

anotadas con **@Configuration**, donde se ejecutan todos los métodos anotados con **@Autowired**, recibiendo la inyección de los parametros definidos.

```
@Bean
public PasswordEncoder passwordEncoder(){
    PasswordEncoder encoder = new BCryptPasswordEncoder();
    return encoder;
}

@Autowired
public void configureGlobalSecurity(AuthenticationManagerBuilder auth, PasswordEncoder
pe) throws Exception {
    auth.userDetailsService(userDetailsService()).passwordEncoder(pe);
}

public UserDetailsService userDetailsService(){
    Properties usuarios = new Properties();
    usuarios.put("Fernando",
"$2a$10$SMPYtil7Hs2.cV7nrMjrM.dRAkuoLdYM8NdVrF.GeHfs/MrzcQ/zi,ROLE_AGENTE,enabled");
    usuarios.put("Mulder"  ,
"$2a$10$M2JRRHUHTfv4uMR4NWmCLebk1r9DyWSwCMZmuq4LKbImOkfhGFAIa,ROLE_AGENTE_ESPECIAL,enable
d");
    usuarios.put("Scully"  ,
"$2a$10$cbF5xp0grCOGcI6jZvPhA.asgmILATW1hNbM2MEqGJEFnRhhQd3ba,ROLE_AGENTE_ESPECIAL,enable
d");
    usuarios.put("Skinner" ,
"$2a$10$ZFtPIULMcxPe3r/5VunbVujMD7Lw8hkqAmJlxmK5Y1TK3L1bf8ULG,ROLE_DIRECTOR,enabled");

    return new InMemoryUserDetailsManager(usuarios);
}
```

NOTE	Para la configuración anterior, no será necesario definir el método **configure(AuthenticationManagerBuilder auth)**, ya que puede ser sustituido por este otro.

Remember Me

Funcionalidad que permite incluir una **Cookie** para que la aplicación no pida al usuario que realice el proceso de **login**, recordandolo.

```
http
    .rememberMe()
        .rememberMeParameter("remember-me-param")
        .rememberMeCookieName("my-remember-me")
        .tokenValiditySeconds(86400);
```

Seguridad en la capa transporte - HTTPS

Se puede indicar a Spring que peticiones necesitan de un canal seguro (https), así como establecer redirecciones automaticas entre puertos.

```
http
    .requiresChannel()
        .anyRequest().requiresSecure()
.and()
    .portMapper()
        .http(8080).mapsTo(8443);
```

Para activar HTTPS, se ha de realizar configuraciones en el servidor, en el caso de un tomcat, se haria algo así.

En el fichero de configuración **server.xml** *copiar*

```
<Connector SSLEnabled="true" acceptCount="100"
    connectionTimeout="20000" executor="tomcatThreadPool"
    keyAlias="tcserver" keystoreFile="${catalina.base}/conf/tcserver.keystore"
    keystorePass="changeme"
    maxKeepAliveRequests="15" port="8443" protocol="HTTP/1.1"
    redirectPort="8443" scheme="https" secure="true"/>
```

Sesiones concurrentes

En el directorio **${catalina.base}/conf** *copiar un fichero de claves* **tcserver.keystore** *con los datos definidos en la configuración anterior.*

Se puede controlar la concurrencia de sesiones, permitiendo controlar que varios navegadores accedan con el mismo usuario. El procedimiento crea un contador, que cuando cumple con el numero especificado, no deja crear nuevas conexiones.

```
http
    .sessionManagement()
        .maximumSessions(1)
        .maxSessionsPreventsLogin(true);
```

Tiene un problema cuando se cierra el navegador y no se da a desconectar, ya que no se ejecuta la actualización del contador, luego cuando se llegue al máximo ya solo se podría acceder desde los navegadores que consiguieron el acceso.

Para solventar este problema, se dipone de un Listener que se encarga de está eventualidad.

```
public class AppInitializer implements WebApplicationInitializer {

    @Override
    public void onStartup(ServletContext servletContext) throws ServletException {
        servletContext.addListener(new HttpSessionEventPublisher());
    }
}
```

SessionFixation

Es un efecto que se puede producir en las aplicaciones web, que consiste en la sobrescritura de los roles en el objeto sesión.

La idea basicamente es que al realizarse el proceso de **login** sobre una sesión ya creada (repetir el login), sino se crea de nuevo un objeto session con su correspondiente identificador de sesión (JSESSIONID), es decir se recicla el objeto sesión, se puede producir que existan dos usuarios con la misma sesión, pero con los datos de roles del segundo usuario, que puede tener mas permisos.

Spring por defecto lo contempla.

```
http
    .sessionManagement()
        .sessionFixation()
        .migrateSession();
```

A mayores, se puede proporcionar una migración de los datos de la sesión antigua a la nueva, para poder seguir manteniendo información aunque se produzca el cambio de usuario.

Libreria de etiquetas

Para añadir las etiquetas propias de la seguridad se ha de incluir la cabecerá

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

Permite el acceso al objeto de authenticación para

1. Controlar que partes de la **View** se van a renderizar

```
<sec:authorize access="hasRole('ROLE_DIRECTOR')">
    <a href="<c:url value='/expedientesx/clasificar?id=${expediente.id}'/>">
clasificar</a>
    <a href="<c:url value='/expedientesx/desclasificar?id=${expediente.id}'/>"
>desclasificar</a>
</sec:authorize>
```

NOTE Este tipo de seguridad, unicamente protege que en el uso normal de la aplicación un usuario pueda ver un enlace o información, a la que no tiene acceso, para evitar errores de acceso, pero no protege la funcionalidad (negocio) en si.

1. Mostrar información del usuario

```
<sec:authentication property="principal.username" />
```

Expresiones SpEL

Para crear las expresiones que se han ido empleando, el lenguaje de expresiones de Spring (SpEL), ofrece una serie de comandos

- hasRole(role)
- hasAnyRole([role1,role2])
- permitAll
- denyAll
- isAnonymous()
- isAuthenticated()

Seguridad de métodos

Para activar la seguridad de los métodos, se ha de incluir la anotación **@EnableGlobalMethodSecurity** en la clase **@Configuration**

```
@Configuration
@EnableWebSecurity
@EnableGlobalMethodSecurity
public class ConfiguracionSpringSecurity extends WebSecurityConfigurerAdapter {}
```

Tambien es recomendable definir una página de error cuando se produzcan intentos de acceso no autorizados

```
http
    .exceptionHandling()
        .accessDeniedPage("/paginas/acceso-denegado.jsp");
```

Spring Security, es compatible con anotaciones propias y de la especificación java JSR-250. Las propias se dividen en dos grupos

- **@Secured** que se activa con **securedEnabled**
- **prepost** que se activa con **prePostEnabled**

Para activar las de JSR-250 habrá que activar **jsr250Enabled**.

```
@Configuration
@EnableWebSecurity
@EnableGlobalMethodSecurity(securedEnabled = true, jsr250Enabled=true, prePostEnabled =
true)
public class ConfiguracionSpringSecurity extends WebSecurityConfigurerAdapter {}
```

Las anotaciones de Spring son

- **@Secured**

```
@Secured("ROLE_AGENTE_ESPECIAL,ROLE_DIRECTOR")
void clasificar(Expediente expediente);
```

- **@PreAuthorize**

```
@PreAuthorize("hasRole('ROLE_DIRECTOR') or #expediente.investigador ==
authentication.name")
void desclasificar(Expediente expediente);
```

- **@PostAuthorize**

```
@PostAuthorize("hasRole('ROLE_DIRECTOR') or returnObject.investigador ==
authentication.name")
Expediente mostrar(Long id);
```

- **@PreFilter**
- **@PostFilter**

```
@PostFilter("(hasRole('ROLE_AGENTE') and not filterObject.clasificado) " +
    "or (hasAnyRole('ROLE_AGENTE_ESPECIAL','ROLE_DIRECTOR') and not
filterObject.informe.contains(principal.username))")
List<Expediente> listarTodos();
```

Las anotaciones de JSR-250 son

- **@RolesAllowed**

```
@RolesAllowed("ROLE_AGENTE_ESPECIAL,ROLE_DIRECTOR")
void desclasificar(Expediente expediente);
```

- PermitAll
- DenyAll
- RunAs