hiberus
TECNOLOGIA

Curso de introducción a Spring e Hibernate

Septiembre 2017

Crecemos contigo

1. ¿Qué es Spring Framework?

El framework Spring proporciona un modelo de programación y configuración completo para aplicaciones basadas en Java. Este framework se centra en facilitar el desarrollo de aplicaciones, mediante la estandarización de componentes y metodologías para que los equipos de desarrollo puedan centrarse en tareas de producción de más alto nivel.

Spring fue escrito originalmente para la plataforma J2EE de Java, plataforma orientada al desarrollo de aplicaciones web, y ha ido evolucionando rápidamente hasta el día de hoy, donde podemos encontrar diferentes ramas de desarrollo de la mano de SpringSource y todo su equipo de desarrolladores.

1.1. Principales características

Spring Framework es en la actualidad la referencia en el mundo de los frameworks de programación para los desarrolladores web de todo el mundo. Su éxito se fundamenta principalmente en la constante labor de investigación e innovación que realiza su equipo de desarrollo oficial, y de una amplia y activa comunidad de desarrolladores que mejoran y amplían sus funcionalidades.

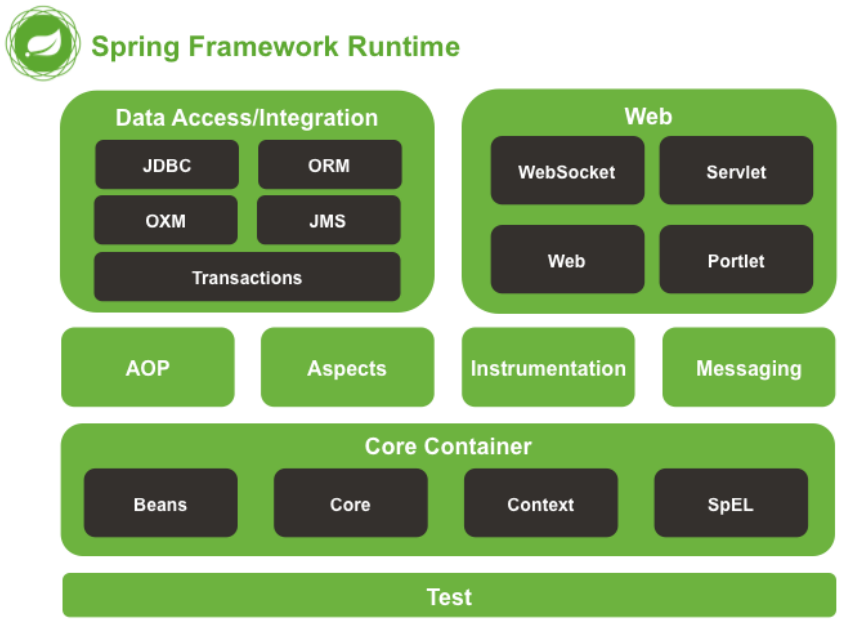
Se trata de un framework que impulsa una metodología de trabajo ágil, eficiente y de buena praxis, lo que resulta en la creación de Software de elevada calidad y mantenibilidad.

Tiene amplia compatibilidad para la integración con otros frameworks y librerías de uso común para la creación de aplicaciones web, desde Composite Views como Velocity o Tiles, APIs en capa de persistencia como Hibernate o JDO, y otra miscelánea de recursos como JavaMail, Quartz, etc…

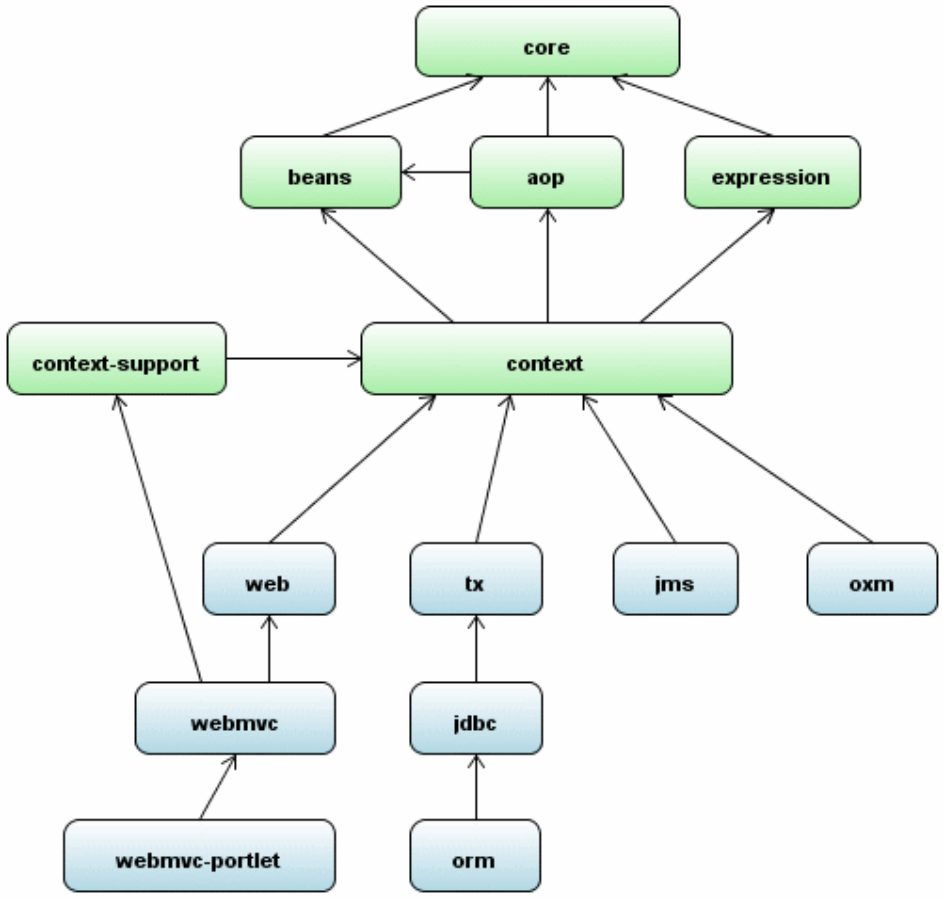
1.2. Módulos

El framework Spring consta de una serie de funcionalidades organizadas en diversos módulos. Estos módulos se agrupan según se muestra en la siguiente figura:

Parque Empresarial PLAZA
C/ Bari 25, Escalera 1 50197 Zaragoza T:+34 902 87 73 92/ F: 976 10 66 21



En el siguiente esquema se pueden ver las dependencias de cada uno de los módulos.



Parque Empresarial PLAZA
C/ Bari 25, Escalera 1 50197 Zaragoza T:+34 902 87 73 92/ F: 976 10 66 21

Core Container → Comprende los módulos spring-core, spring-beans, spring-context, spring-context-support, y spring-expression (Spring Expression Language).

- Los módulos **spring-core** y **spring-beans** proporcionan las partes fundamentales del framework, incluyendo la inyección de dependencias.

- El módulo **spring-context** se construye sobre la sólida base proporcionada por los módulos Core y Beans. Utiliza funciones del módulo Beans para añadir soporte para la internacionalización (utilizando, por ejemplo, paquetes de recursos), la propagación de eventos, la carga de recursos,…

- El módulo **spring-context-support** proporciona soporte para la integración de librerías de terceros en el contexto de aplicación de Spring, para caché (EhCache, Guava, JCache), correo (JavaMail), programación de tareas (CommonJ, Quartz) y motores de plantillas (FreeMarker, JasperReports, Velocity).

- El módulo **spring-expression** proporciona un potente lenguaje de expresiones para consultar y manipular objetos en tiempo de ejecución.

Web → Comprende los módulos spring-web, spring-webmvc, spring-websocket y spring-webmvc-portlet.

- El módulo **spring-web** ofrece funciones de integración web básicas, como la función de carga de archivos en varias partes o la inicialización del contenedor IoC.

- El módulo **spring-webmvc** provee el modelo-vista-controlador de Spring y el soporte para la implementación de los servicios web para aplicaciones web.

- El módulo **spring-websocket** contiene las funciones que dan soporte a la tecnología websocket.

- El módulo **spring-webmvc-portlet** proporciona la implementación MVC para ser utilizado en un entorno de portlets.

Data Access/Integration → Comprende JDBC, ORM, OXM, JMS y módulos de transacción.

- El módulo **spring-jdbc** contiene una abstracción de la capa del JDBC.

- El módulo **spring-tx** proporciona soporte básico para transacciones.

Parque Empresarial PLAZA
C/ Bari 25, Escalera 1 50197 Zaragoza T:+34 902 87 73 92/ F: 976 10 66 21

- El módulo **spring-orm** proporciona la capa de integración con las APIs de mapeo objeto-relacional más populares, como Hibernate, JDO, Ibatis, etc…

- El módulo **spring-oxm** proporciona soporte para la integración con manejadores XML como JAXB, Castor, XMLBeans, etc…

- El módulo **spring-jms** contiene las funciones para consumir y producir mensajes.

Test

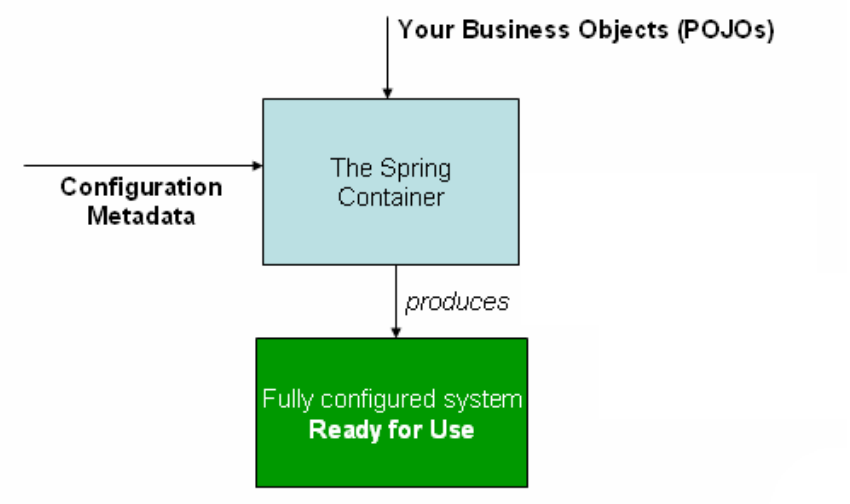
- El modulo **spring-test** da soporte para realizar pruebas unitarias y pruebas de integración de componentes de Spring con JUnit o TestNG.

Parque Empresarial PLAZA
C/ Bari 25, Escalera 1 50197 Zaragoza T:+34 902 87 73 92/ F: 976 10 66 21

2. Application Context

El contexto de aplicación de Spring, es el lugar donde se guardan todas las instancias de los beans que forman parte de la aplicación.

Los beans, son declarados a partir de una serie de metadatos (por xml o anotaciones Java) y clases java que sirven a Spring para generar las instancias en el Application Context.



Podemos entender el Application Context, como una sección de memoria dentro del contenedor de aplicaciones utilizada como almacén de instancias de objetos inicializados y configurados que darán lugar a todas las funcionalidades de la aplicación.

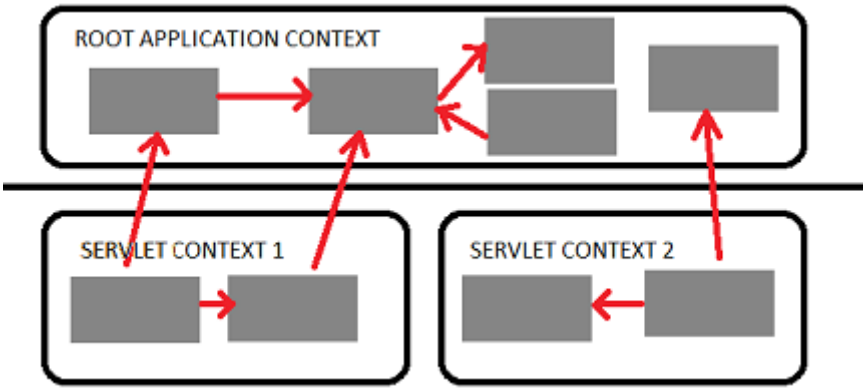
El Application Context es definido por una interface, del mismo nombre (ApplicationContext) que dispone de varias extensiones e implementaciones. La extensión más utilizada, especialmente en entornos de servicios web, es la WebApplicationContext, que acaba siendo implementada por AnnotationConfigWebApplicationContext, o XmlWebApplicationContext entre otras.

Estas dos implementaciones, crean un Application Context compatible con una serie de características estándar para entornos web, como la resolución de mensajes estáticos, internacionalización o la implementación de un evento entre los componentes de la aplicación.

2.1. Jerarquía de contextos

Realmente, un Application Context, puede ser más complejo que un único espacio donde se almacenan todos los beans, generando jerarquías de contextos padre-hijo que limitan el alcance y la visibilidad de los mismos.

Parque Empresarial PLAZA
C/ Bari 25, Escalera 1 50197 Zaragoza T:+34 902 87 73 92/ F: 976 10 66 21



Como se puede ver en la imagen, el Application Context general, expone sus beans al resto de beans de su contexto y a los contextos hijos, pero los beans de los contextos hijos, solo son visibles por los beans de su propio contexto.

Esta característica, permite a los desarrollares aislar diferentes beans de la aplicación, unos de otros, y al mismo tiempo, configurar otros como globales, visibles y accesibles por el resto de beans de la aplicación.

Esto tiene sentido a la hora de tratar con HttpServlets, y las particularidades que cada uno de ellos puedan tener.

Por ejemplo, podemos tener un servidor, que cuente con una serie de interfaces públicas para renderizar aplicaciones web, y otras para dar soporte a clientes mediante una API Rest.

Aunque Spring, permite gestionar interfaces de forma sencilla, independientemente de la configuración del Servlet (de hecho utiliza la misma implementación en los dos casos), podemos querer contar con controladores específicos, una capa de seguridad completamente diferente para cada uno de los entornos, etc...

Parque Empresarial PLAZA
C/ Bari 25, Escalera 1 50197 Zaragoza T:+34 902 87 73 92/ F: 976 10 66 21

3. BEANS

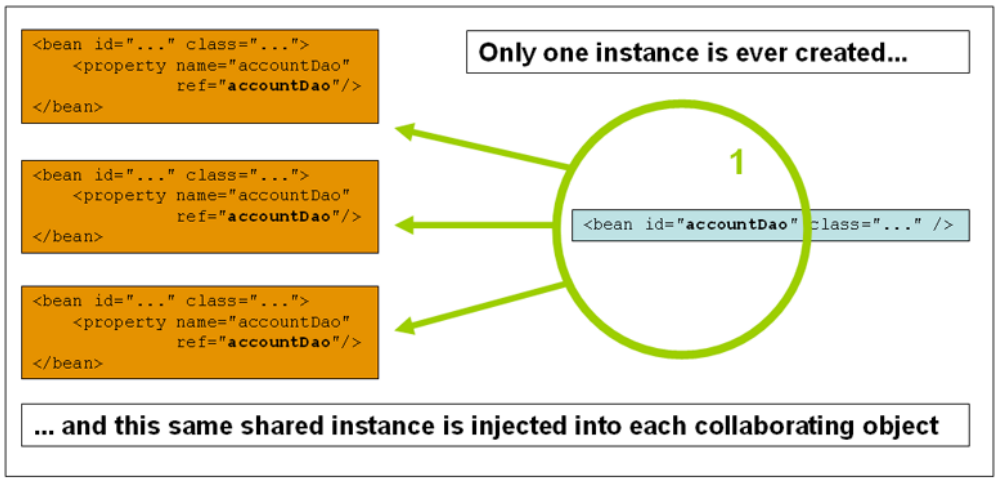
Los beans en Spring son considerados recipes, es decir, son una simple estructura, marcada desde un xml o anotación para ser creada mediante diferentes estrategias como un bean del Spring Application Context. Esto quiere decir, que utilizando una referencia del ApplicationContext o BeanFactory, podemos evitar las estrategias proveídas por el framework y crear nuestros propios beans manualmente, lidiando, eso sí, con el orden y dependencias de estos beans por nosotros mismos.

Pero, aparte del concepto de Class, Recipe (clase marcada para ser un Bean), y Bean, instancia de un recipe (o no) en el application context, está el concepto de Scope.

3.1. Bean Scope

Spring provee cinco tipos de Scope, tres de ellos solo habilitados desde un web-aware ApplicationContext, por ejemplo WebApplicationContext, que extiende desde el modulo spring-web la interface ApplicationContext.

Singleton (scope por defecto) → Una instancia de Bean por Spring Container.

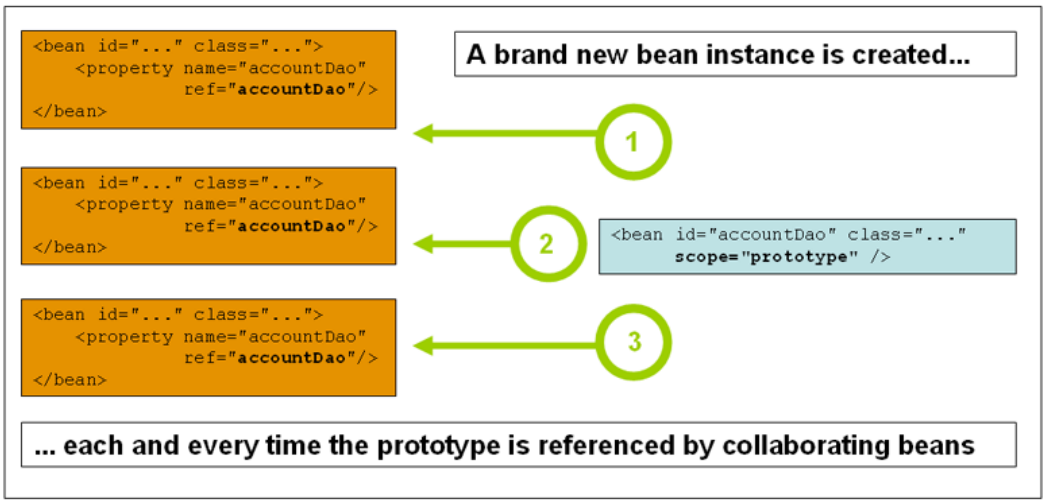


Algunos aspectos clave, es que el termino Singleton difiere del patrón de diseño convencional, no se trata de asegurar una instancia por tipo de objeto, sino una instancia por tipo de objeto e id en un mismo Spring Container.

Además, estos Beans no son thread safe, por lo tanto, cualquier atributo o referencia debe ser tratada de forma manual y asegurar su correcto funcionamiento en un entorno de ejecuciones concurrentes.

Parque Empresarial PLAZA
C/ Bari 25, Escalera 1 50197 Zaragoza T:+34 902 87 73 92/ F: 976 10 66 21

Prototype → Una nueva instancia de Bean cada vez que la IoC lo resuelve como referencia.



Request → Una nueva instancia de bean por client-request. Los cambios sobre estos Beans no son visibles por ninguno otro, cuando la request termina, el bean se destruye.

Session → Una nueva instancia de bean por client-session. Los cambios sobre estos Beans son visibles en las sucesivas request realizadas en la misma client-session.

Application → El comportamiento es similar a Singleton, pero difiere en que la instancia de estos Beans es almacenada como atributo del *ServletContext*, no de un Spring Container. Esto supone diferencias significativas, como por ejemplo, poder acceder a dichas instancias desde un JSP.

```
${applicationScope['myApplicationBean.myApplicationBeanAttribute']}
```

3.2. Ciclo de vida en Spring

Cuando una aplicación con Spring es inicializada por primera vez, un orden y una serie de mecanismos de inicialización son utilizados para instanciar y aplicar la IoC a los beans que formarán parte del application context.

Por un lado, está el orden lógico de inicialización. Este orden, es el aplicado por Spring para ir resolviendo las diferentes dependencias entre beans.

SI A depende de B, B será instanciado en primer lugar. Esto ocurre en todos los casos salvo cuando no se trata de referencias fuertes, en cuyo caso, podemos especificar al BeanFactory que existe una dependencia débil entre ciertos beans, utilizando la propiedad depends-on, o @DependsOn en anotación. Si un bean es instanciado desde xml, @DependsOn es ignorado.

Parque Empresarial PLAZA
C/ Bari 25, Escalera 1 50197 Zaragoza T:+34 902 87 73 92/ F: 976 10 66 21

Por otro lado, están las estrategias de inicialización, Spring provee mecanismos para inicializar beans.

- **@PostConstruct**, es una anotación que permite especificar que un método será ejecutado después de ser instanciado, **@PreDestroy** establece que un método será ejecutado justo antes de ser destruido.
- Trabajando con xml, mediante la propiedad **init-method** o **destroy-method**.
- La interface **InitializingBean**, también permite la inicialización de un objeto implementando el método *afterPropertiesSet()*.
- La interface **DisposableBean**, permite la ejecución del método *destroy()* antes de que el bean sea destruido.

Siendo @PostConstruct y @PreDestroy las estrategias de post inicialización y pre destrucción más extendidas y recomendadas. Si se combinan diferentes estrategias, se tiene que tener en cuenta el orden en el que Spring las resuelve.

Métodos de inicialización:

@PostConstruct → InitializingBean → init-method

Métodos de destrucción:

@PreDestroy → DisposableBean → pre-destroy

Algunas consideraciones a tener en cuenta, es que determinados decorators de Spring todavía no han sido configurados cuando los métodos de inicialización han sido invocados, provocando ciertas incompatibilidades.

Por ejemplo, utilizar @Transactional en un método anotado con @PostConstruct no se resolverá de forma correcta, y todo el código ejecutado en @PostConstruct será ejecutado fuera de un entorno transaccional, la manera de resolver esta incompatibilidad, es inyectar como dependencia fuerte PlatformTransactionManager y utilizar su referencia en el método.

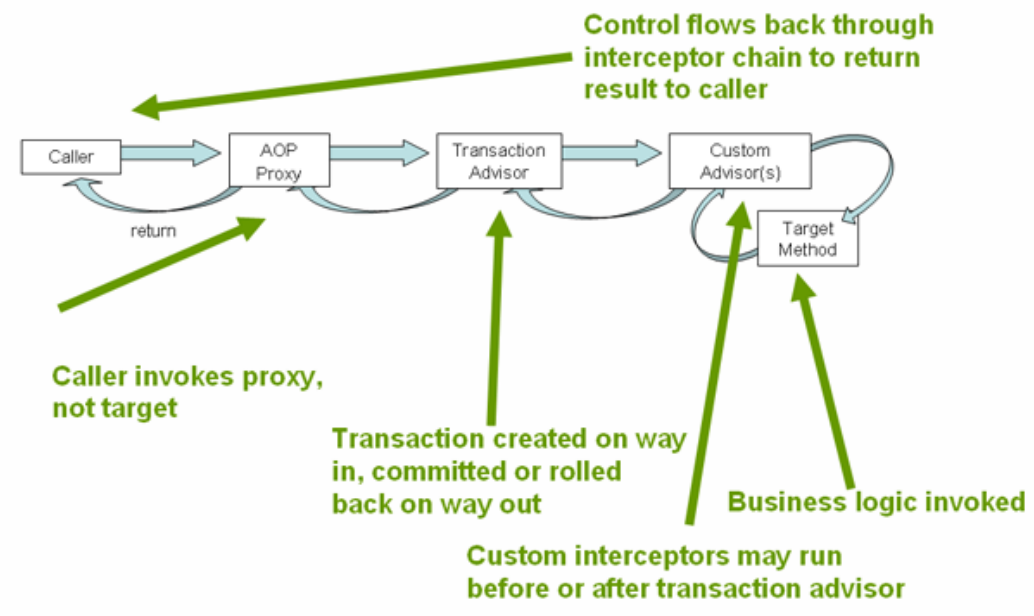
3.3. Bean proxies

Aunque siempre nos refiramos a los beans de Spring como instancias de objetos configurados en el Application Context, realmente esto no es así, Spring, utiliza diferentes técnicas para automatizar procesos, integrando AOP (spring-aop y spring-aspects) con java proxies.

Estos últimos, encapsulan, a modo de decorators, las instancias de los objetos creados a partir de nuestros recipe beans añadiéndoles funcionalidades.

Parque Empresarial PLAZA
C/ Bari 25, Escalera 1 50197 Zaragoza T:+34 902 87 73 92/ F: 976 10 66 21

Por ejemplo, una instancia de un @Transactional @Service, acaba instanciándose de la siguiente manera en el Application Context.



Al final de toda la cadena de decorators (en Target Method) es donde se encuentra la lógica de negocio implementada por el programador.

Fuente de errores y limitaciones:

Estas técnicas abstraen notablemente al programador de lo que está sucediendo realmente antes y después de la invocación de su método, y a pesar de que Spring ofrece muchas posibilidades de configuración en sus abstracciones, en esta técnica, existe una particularidad que es fuente de muchos errores a nivel de arquitectura y de implementación por parte de los desarrolladores.

La limitación tiene su origen en la naturaleza de los java proxies, ya sea en cualquiera de las dos implementaciones soportadas por Spring (JDK dynamic proxies o CGLib proxies) estos proxies actúan como wrappers, y por tanto solo son conscientes de lo que ocurre en llamadas realizadas desde fuera del wrapper hacia el contenido de su interior.

Por ello, todas las inner calls son desconocidas por estos proxies, y como resultado, cuando un método realiza una inner call a otro método de la misma clase, en el que por ejemplo, se redefine la propagación del ámbito @Transactional, este no es reconocido y no es tenido en cuenta por los decorators, como resultado, la propagación no se redefine.

Parque Empresarial PLAZA
C/ Bari 25, Escalera 1 50197 Zaragoza T:+34 902 87 73 92/ F: 976 10 66 21

Para evitar esto, existen varias técnicas:

- **Redefinir la arquitectura** → De tal manera que nos aseguremos que para una funcionalidad concreta, allá donde haya un decorator que redefina cualquier comportamiento a desarrollar por algún decorator, siempre se realiza mediante una outer call a otro bean.

- **Workaround** → Es una técnica, por la que un objeto, obtiene del ApplicationContext una copia de su propia instancia, llamándose a sí mismo y asegurándose de que la llamada atraviesa el java proxy.

- **Aspect weaving** → Una técnica implementada por la librería aspectweaver, la cual, sirviéndose de un ClassLoader modificado es capaz de redefinir los encaminamientos de código estándar, eliminando la li-mitación de las inner calls.

3.4. Sobrescribir beans

A la hora de sobrescribir un bean hay que tener en cuenta varias cosas. Primero, si realmente queremos sobrescribirlo o solo extenderlo. Al extenderlo se crea un nuevo bean, partiendo de la clase que se extiende, pero el bean de la clase que padre sigue existiendo en el aplication context.

```
@Service
public class BeanServicePr1extImpl extends BeanServicePr1Impl {

    …

}
```

Ahora tenemos dos beans (BeanServicePr1Impl, BeanServicePr1extImpl). Lo que nos va a suponer un problema a la hora de inyectarlos. Si, al inyectar estos componentes, solo usamos @Autowired, Spring nos va a mostrar un error porque existen dos beans del mismo tipo. Con lo que debemos especificar cuál de los dos queremos inyectar. En el application context todos los beans son almacenados asociados a una clave, por defecto el nombre de la clase.

```
@Autowired
@Qualifier("beanServicePr1Impl")
private BeanServicePr1 beanService;
```

De esta forma le indicamos cuál de los dos beans queremos inyectar.

Pero qué ocurre si lo que queremos es tener solo uno de estos dos beans, el extendido, y que allí donde se estuviera usando el padre pase a usarse nuestro nuevo bean. Para esto tenemos que hacer que cuando Spring inicialice los beans no inicialice el que vamos a extender.

Parque Empresarial PLAZA
C/ Bari 25, Escalera 1 50197 Zaragoza T:+34 902 87 73 92/ F: 976 10 66 21

Esto se le informa en la clase de configuración ApplicationConfig (o en el fichero xml correspondiente).

```
@Configuration
@PropertySource("classpath:application.properties")
@ComponentScan(
        basePackages = "com.hiberus.api.service",
        excludeFilters = {
                @ComponentScan.Filter(type = FilterType.ASPECTJ, pattern =
                                                "com.hiberus.api.service.practice1.impl.BeanServicePr1Impl")})
public class ApplicationConfig {

   …

}
```

De esta forma se le indica a Spring que en esa clase no busque un bean, con lo que no lo inicializará. Como resultado obtenemos un nuevo bean, que extiende del anterior, y que, al haber excluido al padre, el nuevo ocupará su lugar cuando sea inyectado.

Cómo se comentaba antes, en el application context, todos los beans son almacenados asociados a una clave, por defecto el nombre de la clase. Si en algún lugar del código se hubiera inyectado el padre utilizando la anotación @Qualifier, al intentar inyectar en su lugar al hijo, Spring devolvería un error, puesto que como las clases tienen nombres diferentes no podría encontrarlo. La forma de solucionar esto es que el nuevo bean, aparte de extender al anterior, tome también su nombre.

```
@Service("beanServicePr1Impl")
public class BeanServicePr1extImpl extends BeanServicePr1Impl {

    …

}
```


Parque Empresarial PLAZA
C/ Bari 25, Escalera 1 50197 Zaragoza T:+34 902 87 73 92/ F: 976 10 66 21

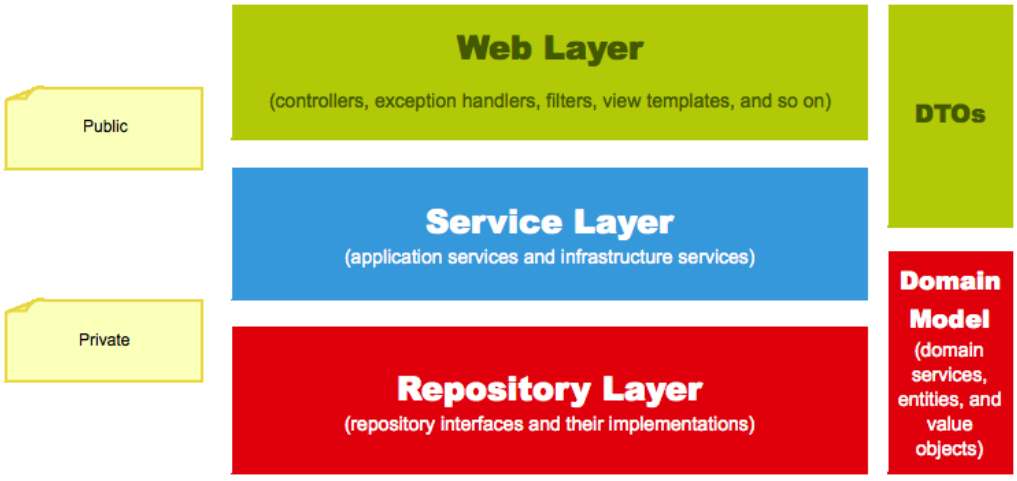
4. Arquitectura de un proyecto con Spring

Cuando pensamos inicialmente en la creación de un nuevo proyecto tenemos que tener claro cuál va a ser la función que van a tener las instancias de nuestros objetos, según su naturaleza, van a formar parte de una capa u otra en una arquitectura de software basada en capas.

Como normal general, cuando programamos aplicaciones con Spring, hablamos de arquitectura Model-View-Controller, de hecho, Spring, facilita en gran medida con sus implementaciones básicas su implementación, guiando a los desarrolladores a realizar una arquitectura MVC lo más lógica posible.

Por ejemplo, desde Spring se facilita un conjunto de anotaciones como @Controller, @Service y @Repository con el mismo comportamiento, ya que todas extienden a @Component, sin embargo, tienen un nombre con una semántica completamente diferente, que ayuda a los desarrolladores a diferenciar a que capa de arquitectura pertenece el recipe bean del que forman parte.

@Component notifica al annotation-driven que se trata de un recipe bean que debe ser instanciado, las implementaciones de @Component facilitan una semantica visual al desarrollador para entender a primera vista, a que capa de arquitectura pertenece dicho recipe.



4.1. Arquitectura MVC

Una vez que tenemos claro los tres niveles de una arquitectura con Spring, queda la parte esencial, entender el ámbito y las responsabilidades de los beans que forman parte de cada capa.

Parque Empresarial PLAZA
C/ Bari 25, Escalera 1 50197 Zaragoza T:+34 902 87 73 92/ F: 976 10 66 21

Controller

Los Controllers son beans directamente relacionados con el DispatcherServlet. Este Servlet, es una implementación de Spring con muchas funcionalidades añadidas sobre un Servlet estándar de JavaEE. Una de sus principales funcionalidades es la de implementar de forma automática un Front-Controller.

Spring, reconoce automáticamente mediante la utilización de expresiones regulares la estructura de una url, y la compara con las interfaces creadas mediante la anotación @RequestMapping en cada uno de los métodos públicos de los recipe beans anotados con @Controller. A partir de ahí, delega la request en dicho método, el cual debería de llamar a una, y solo una, interface pública de cualquier de los @Service beans de nuestra aplicación.

Este último punto es crucial en una correcta comprensión de una arquitectura MVC, un @Controller no debería realizar ninguna operación de negocio, y debería mitigar lo máximo posible cualquier operación lógica que se realice sobre él.

Las competencias de un @Controller son claras:

- Relacionar una interface pública (generalmente una url) invocada desde un cliente, y gestionada por el DispatcherServlet hasta un método del @Controller, con una interface publica de un @Service de nuestra aplicación.
- Gestionar la validación de los datos recibidos.
- Manejar las excepciones generadas en el workflow.

Lo que no debe de hacer un @Controller:

- No debe realizar ninguna operación de negocio.
- No debe invocar, en la medida de lo posible, más de una interface publica de un @Service.
- No debe interaccionar con objetos del modelo, excepto en caso de validaciones.

La correcta implementación de un @Controller va a adquirir a medio-largo plazo una importancia vital en el mantenimiento de nuestros proyectos.

Para tener una visión más clara de porque esto es así, podemos pensar en dos conceptos claros de programación. Redundancia y transaccionalidad.

Tener lógica de negocio en un controlador, en última instancia, supone no poder disponer de esa lógica en las interfaces públicas de nuestra capa de servicios, impidiendo su reutilización y fomentando la proliferación de código duplicado.

A nivel transaccional, la importancia es mucho mayor, ya que un @Controller debe permanecer ajeno a cualquier ámbito transaccional, y por tanto, se altera un workflow transaccional único, simple, así como la gestión de rollbacks.

Parque Empresarial PLAZA
C/ Bari 25, Escalera 1 50197 Zaragoza T:+34 902 87 73 92/ F: 976 10 66 21

Service

Es la capa encargada de gestionar las operaciones lógicas a realizar, puede alterar información, tomar decisiones dependiendo de ella, conoce los recursos, bases de datos y funcionalidades a desarrollar por las diferentes acciones a petición del cliente.

Un error muy común en arquitecturas Spring, es no tener claro este concepto, y acabar confundiendo en muchos casos esta capa @Service como un mero enlace entre un @Controller y un @Repository.

Los @Service son capaces de tomar decisiones complejas, delegando responsabilidades en otros métodos de sí mismos o incluso de otros @Service.

Repository

Es la capa encargada de gestionar el almacenamiento de información en diferentes repositorios (orígenes de datos), puede ser desde un DBMS como MySQL, SQLServer, Oracle... un repositorio orientado a documentos, como Elastic, OrientDB... o incluso un Datagrid de memoria distribuida, como Hazelcast, Redis, Memcached,...

Los beans de esta capa, por norma general, suelen tener un comportamiento genérico muy habitual, casi siempre son básicas operaciones CRUD gestionadas por librerías de terceros que permiten la utilización de lógica en operaciones similares, aplicadas sobre entidades de persistencia diferentes modeladas mediante objetos Java.

Por ello es buena idea considerar la implementación de una capa genérica para las operaciones más comunes, que en última instancia pueda ser extendida para operaciones concretas de cada una de las entidades.

Parque Empresarial PLAZA
C/ Bari 25, Escalera 1 50197 Zaragoza T:+34 902 87 73 92/ F: 976 10 66 21

5. Configuración de un proyecto

Todo lo que ocurre alrededor de una aplicación desarrollada con Spring gira en torno a la configuración de su contexto de aplicación, aquí, es donde se configuran los diferentes componentes de la aplicación que van a desarrollar diferentes funcionalidades.

Al principio, puede resultar muy confuso llegar a tener una idea general del framework, pero al final, son un conjunto de filtros, servlets y componentes (beans) configurados en el application context.

5.1. Ficheros de configuración

En toda configuración de un proyecto con Spring MVC existen 3 ficheros principales de configuración.

- **application.xml** (ApplicationConfig): Es el encargado de definir los beans que van a ser instanciados y almacenados en el ApplicationContext. En el caso de ApplicationConfig, es la clase de configuración principal, importa el resto de clases de configuración que definen el RootApplicationContext. Es la clase que deberá indicarse como RootConfig en el WebAppInitiializer.
- **dispatcher-servlet.xml** (DispatcherServletConfig): Se encarga de definir los controladores que serán instanciados y de definir el http servlet que recibirá las peticiones y las encaminará hacia el controlador correspondiente.
- **web.xml** (WebAppInitializer): Es el fichero principal. En él se definen los ficheros de configuración, tanto el application.xml (y otros ficheros de configuración que pueda haber) como el dispatcher-servlet.xml y los url-pattern que atenderá.

ApplicationConfig

XML:

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xsi:schemaLocation="http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans.xsd">

    <bean id="catalogAdministratorService"
          class="org.springframework.transaction.interceptor.TransactionProxyFactoryBean">
        <property name="transactionManager">
            <ref bean="transactionManager" />
        </property>
        <property name="target">
            <ref bean="catalogAdministratorServiceTarget" />
        </property>
```


Parque Empresarial PLAZA
C/ Bari 25, Escalera 1 50197 Zaragoza T:+34 902 87 73 92/ F: 976 10 66 21

```
        <property name="transactionAttributes">
            <props>
                <prop key="getFullProductsToIndex">PROPAGATION_REQUIRES_NEW, readOnly</prop>
                <prop key="get*">PROPAGATION_REQUIRED, readOnly</prop>
                <prop key="*">PROPAGATION_REQUIRED</prop>
            </props>
        </property>
    </bean>

    <bean id="catalogAdministratorServiceTarget"
          class="com.hermes.front.index.service.impl.CatalogAdministratorServiceImpl">
        <property name="VProductoAtributoDAO"><ref bean="vProductoAtributoDAOdeprecated" /></property>
        <property name="productoDAO"><ref bean="productoDAOdeprecated" /></property>
        <property name="vistaUrlRealDAO"><ref bean="vUrlRealDAOdeprecated" /></property>
        <property name="marcaDAO"><ref bean="marcaDAOdeprecated" /></property>
        <property name="idiomaDAO"><ref bean="idiomaDAOdeprecated" /></property>
        <property name="categoriaEcommerceDAO"><ref bean="categoriaEcommerceDAOdeprecated" /></property>
        <property name="vCategoriaEcommerceTraduccionDAO"><ref
bean="vCategoriaEcommerceTraduccionDAOdeprecated" /></property>
        <property name="categoriaEcommerceAtributoValorDAO"><ref
bean="categoriaEcommerceAtributoValorDAOdeprecated" /></property>
        <property name="contenidoDAO"><ref bean="contenidoDAOdeprecated" /></property>
        <property name="traduccionDAO"><ref bean="traduccionDAOdeprecated" /></property>
        <property name="contenidoAtributoDAO"><ref bean="contenidoAtributoDAOdeprecated" /></property>
        <property name="VContenidoAtributoTraduccionDAO"><ref
bean="vContenidoAtributoTraduccionDAOdeprecated" /></property>
        <property name="VContenidoAtributoDAO"><ref bean="vContenidoAtributoDAOdeprecated" /></property>
        <property name="vIdiomaDAO"><ref bean="vIdiomaDAOdeprecated" /></property>
        <property name="buscadorSinonimoDAO"><ref bean="buscadorSinonimoDAOdeprecated" /></property>
        <property name="siteDAO"><ref bean="siteDAOdeprecated" /></property>
    </bean>

   …

</beans>
```

En este ejemplo se ve cómo definir un bean (`catalogAdministratorServiceTarget`) inyectándole dependencias. También se ve cómo definir un proxy en el que se puede especificar la transaccionalidad de cada uno de los métodos del bean al que afecta.

JavaConfig:

```
@Configuration
@ComponentScan(basePackages = "com.hiberus.api.service")
public class ApplicationConfig {

}
```

En este ejemplo se utiliza @ComponentScan (también se puede utilizar en la configuración por xml). Esta anotación indica a Spring a partir de que paquete debe escanear las clases en busca de beans, anotados, que inicializar.

Parque Empresarial PLAZA
C/ Bari 25, Escalera 1 50197 Zaragoza T:+34 902 87 73 92/ F: 976 10 66 21

DispatcherServletConfig

XML:

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

    <context:component-scan base-package="com.hiberus.api.controller" />

    <bean id="viewResolver"
          class="org.springframework.web.servlet.view.UrlBasedViewResolver">
        <property name="viewClass"
                  value="org.springframework.web.servlet.view.JstlView" />
        <property name="prefix" value="/WEB-INF/jsp/" />
        <property name="suffix" value=".jsp" />
    </bean>

</beans>
```

En el ejemplo se utiliza la etiqueta component-scan para indicar el paquete base a partir del cual debe, Spring, buscar clases anotadas.

JavaConfig:

```
@Configuration
@ComponentScan(basePackages = {"com.hiberus.api.controller"})
@EnableWebMvc
public class DispatcherServletConfig extends WebMvcConfigurerAdapter {

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

    @Bean
    public ViewResolver viewResolver() {
        InternalResourceViewResolver viewResolver = new InternalResourceViewResolver();
        viewResolver.setViewClass(JstlView.class);
        viewResolver.setPrefix("/WEB-INF/views/");
        viewResolver.setSuffix(".jsp");

        return viewResolver;
    }

    @Override
    public void configureContentNegotiation(ContentNegotiationConfigurer configurer) {
        configurer.defaultContentType(MediaType.APPLICATION_JSON);
    }

}
```


Parque Empresarial PLAZA
C/ Bari 25, Escalera 1 50197 Zaragoza T:+34 902 87 73 92/ F: 976 10 66 21

WebAppInitializer

XML:

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<web-app xmlns="http://java.sun.com/xml/ns/javaee"
         xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
         xsi:schemaLocation="http://java.sun.com/xml/ns/javaee
                     http://java.sun.com/xml/ns/javaee/web-app_3_0.xsd"
         version="3.0"
         metadata-complete="true">

    <listener>
        <listener-class>org.springframework.web.context.ContextLoaderListener</listener-class>
    </listener>

    <context-param>
        <param-name>contextConfigLocation</param-name>
        <param-value>classpath:/application.xml</param-value>
    </context-param>

    <servlet>
        <servlet-name>dispatcher</servlet-name>
        <servlet-class>org.springframework.web.servlet.DispatcherServlet</servlet-class>
        <load-on-startup>1</load-on-startup>
    </servlet>
    <servlet-mapping>
        <servlet-name>dispatcher</servlet-name>
        <url-pattern>/*</url-pattern>
    </servlet-mapping>

    <welcome-file-list>
        <welcome-file>/login.jsp</welcome-file>
    </welcome-file-list>
</web-app>
```

Como se ha comentado antes, en este fichero se define el application.xml (y todos los que sean necesarios) y el dispatcher servlet. Por defecto, Spring, va a buscar la configuración de cada servlet en un xml de nombre <servlet-name>-servlet.xml.

JavaConfig:

```
public class WebAppInitializer extends AbstractAnnotationConfigDispatcherServletInitializer {

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

        // Don't forget call onStartup parent method to contexts initialization.
        super.onStartup(servletContext);
    }

    @Override
    protected Class<?>[] getRootConfigClasses() {
        return new Class<?>[]{ApplicationConfig.class, RestSecurityConfig.class};
    }

    @Override
    protected Class<?>[] getServletConfigClasses() {
        return new Class<?>[]{DispatcherServletConfig.class};
    }
```


Parque Empresarial PLAZA
C/ Bari 25, Escalera 1 50197 Zaragoza T:+34 902 87 73 92/ F: 976 10 66 21

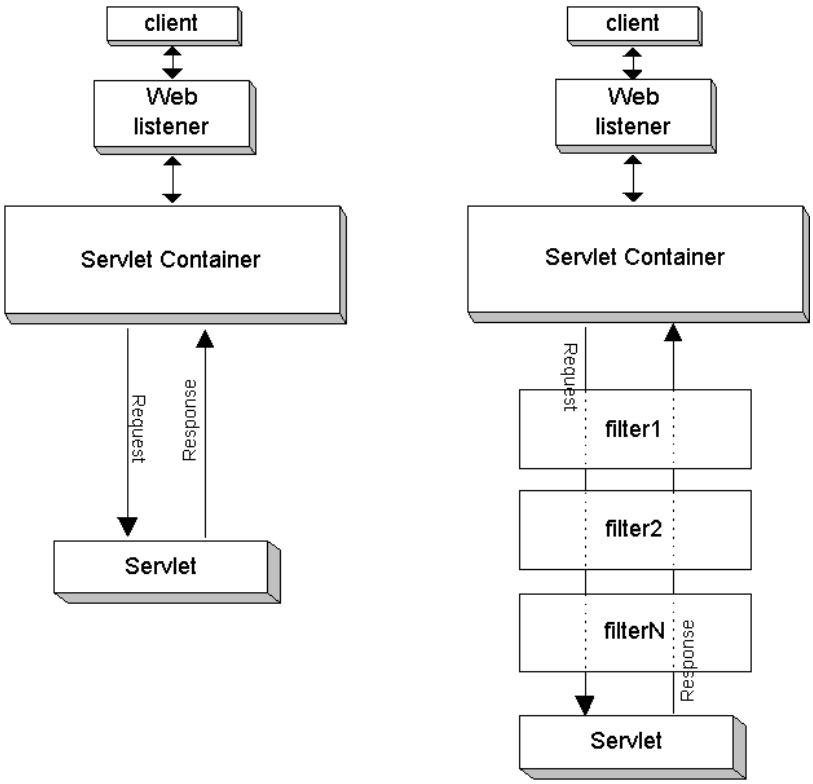
```
    @Override
    protected String[] getServletMappings() {
        return new String[]{"/"};
    }

    @Override
    protected Filter[] getServletFilters() {

        CharacterEncodingFilter characterEncodingFilter = new CharacterEncodingFilter();
        characterEncodingFilter.setEncoding("UTF-8");
        characterEncodingFilter.setForceEncoding(true);

        return new Filter[]{characterEncodingFilter};
    }

}
```

- **getRootConfigClasses** → Debe proporcionar la clase o clases de configuración (@Configuration) donde se define la configuración del ApplicationContext.

- **getServletConfigClasses** → Debe proporcionar la clase o clases de configuración (@Configuration) donde se define la configuración del DispatcherServlet.

- **getServletMappings** → Debe definir la URL o URLS en las cuales se van a mapear los servlets.

- **getServletFilters** → Especificar filtros para las peticiones en el DispatcherServlet. La petición atraviesa los filtros antes de llegar al Servlet correspondiente.



Parque Empresarial PLAZA
C/ Bari 25, Escalera 1 50197 Zaragoza T:+34 902 87 73 92/ F: 976 10 66 21

5.2. Como se inicializa Spring

Cuando el contenedor de aplicaciones inicia el despliegue de la aplicación, busca en el **web.xml** o en la clase **WebApplicationInitializer** (a partir de la especificación Servlet 3.0) la configuración y las directivas para crear el Servlet Context.

Spring, mediante la clase **AbstractAnnotationConfigDispatcherServletInitializer**, que es una implementación compleja de **WebApplicationInitializer** facilita la inicialización de la aplicación, aunque para este documento, durante los siguientes párrafos por cuestiones pedagógicas vamos a hablar de las clases **ContextLoaderListener** y **DispatcherServlet** (abstraída su declaración en el **AbstractAnnotationConfigDispatcherServletInitializer**).

ContextLoaderListener como su propio nombre indica, se trata de un Listener que es declarado en el **web.xml** y cuyo propósito es inicializar un root web context (**WebApplicationContext**), **ContextLoaderListener** por defecto, espera obtener de un parámetro del Servlet Context llamado contextConfigLocation las rutas a los ficheros xml que contienen la definición de beans del Root Context.

DispatcherServlet requiere una explicación más exhaustiva. Lo más destacable, es que no estamos hablando de un Context Listener, sino de un HttpServlet, un tipo de componente de la API JavaEE que sirve para enlazar y manejar peticiones públicas y enviar respuestas.

Spring, desde su módulo web, nos ofrece esta implementación de HttpServlet que maneja mediante beans estándares configurables (sobrescritos realmente) diferentes operaciones comunes en aplicaciones web basadas en JavaEE.

Absolutamente todo lo que ocurre en nuestra aplicación de Spring, pasa por este recurso, ya que es el encargado de procesar las llamadas realizadas a la aplicación y determinar que Controlador será el encargado de atenderla para su resolución.

Este recurso, como hemos dicho antes, se sirve de componentes estándar, como un ViewResolver, que será utilizado por cada Controlador para resolver mediante un ContentNegotiationManager que recurso debe ser enviado como respuesta, desde un documento html, xml, json, pdf, imagen, etc… Un MessageBundle para labores de internacionalización, etc...

Desde la versión 3 de Spring, se utiliza Java Annotations para configurar los recursos de Spring, esta técnica, como otras muchas características del proyecto, dependen de DispatcherServlet.

Los beans objeto de ser utilizados únicamente por cada HttpServlet, se deben crear en un contexto de aplicación propio y únicamente accesible por él, por cuestiones de arquitectura y seguridad, especialmente si se trata de una aplicación con varios HttpServlets.

Parque Empresarial PLAZA
C/ Bari 25, Escalera 1 50197 Zaragoza T:+34 902 87 73 92/ F: 976 10 66 21

Esta diferencia es fundamental para resolver la IoC, puesto que el ámbito de visibilidad solo tiene un camino (hijo → padre), de tal manera que un bean creado en el root ServletContext, no puede resolver una dependencia de otro bean creado en el Contexto de un Servlet (como por ejemplo el del DispatcherServlet).

Parque Empresarial PLAZA
C/ Bari 25, Escalera 1 50197 Zaragoza T:+34 902 87 73 92/ F: 976 10 66 21

6. Uso de controllers y vistas

6.1. HttpMethod

En una API REST, al invocar a un servicio debe indicarse mediante que HttpMethod va a ser invocado. A la hora de definir con Spring MVC un endpoint en un controlador lo mismo, debemos definir para que HttpMethod va a ser publicado.

En Spring disponemos de un enum que contiene todos los métodos disponibles, siendo los más usados GET, POST, PUT y DELETE.

6.2. Definir un controller

Las clases anotadas para ser controladores deben estar ubicadas dentro del paquete o paquetes que se le indicaron al DispatcherServlet en el component-scan. De esta forma podrá encontrarlos e instanciarlos.

A la hora de definir un controlador se debe tener en cuenta sobre que ruta va a actuar. Esta ruta se define con la anotación @RequestMapping.

```
@Controller("webCustomerController")
@RequestMapping("/customer")
public class WebCustomerController {

    …

}
```

Con esto ya tenemos un controlador escuchando en la ruta ‘/custotmer’, pero aún no tenemos nada que se ejecute. Hay que implementar las funciones que responderán a la request. Estas funciones pueden ejecutarse directamente al llamar a la url ‘/customer’ o a una subruta.

```
@RequestMapping(method = RequestMethod.GET)
public String get(Model model) throws Exception {
    …
}

@RequestMapping(method = RequestMethod.PUT)
public String get(Model model) throws Exception {
    …
}

@RequestMapping(value = "/all", method = RequestMethod.GET)
public String getAll(Model model) throws Exception {
    …
}
```


Parque Empresarial PLAZA
C/ Bari 25, Escalera 1 50197 Zaragoza T:+34 902 87 73 92/ F: 976 10 66 21

La primera de las funciones se ejecutará al invocar mediante el método GET a la url '/customer'. La segunda también se ejecutará al invocar a la url '/customer' pero cuando se haga por el método PUT. Mientras que la tercera se lanzará al invocar la url '/customer/all' por el método GET.

Recibir parámetros

Es muy común el hecho de tener que enviar junto a la petición ciertos parámetros para que el servidor actué de una forma u otra, o almacene cierta información. Hay diferentes formas de hacerlo.

- Parámetros de tipo GET:
  Estos parámetros se informan al final de la url (ej. '/customer?name=pepe). Para recuperar este tipo de información podemos hacerlo de dos formas.

```
@RequestMapping(method = RequestMethod.GET)
public String get(@RequestParam("name") String name) throws Exception {
    …
}
```

Se indica como parámetro de la función con la anotación @RequestParam. Esta anotación permite indicar si el parámetro es opcional u obligatorio. En caso de que no venga informado y sea obligatorio (valor por defecto) provocaría una eexcepción.

```
@RequestMapping(method = RequestMethod.GET)
public String get(HttpServletRequest request) throws Exception {
    String name = request.getParameter("name");
    …
}
```

Spring nos permite recuperar la request, y de esta podemos obtener cierta información, como los parámetros.

- Parámetros de la url:
  Estos parámetros se informan como parte de la ruta que se envía al servidor (ej. '/customer/1').

```
@RequestMapping(value = "/{id}", method = RequestMethod.GET)
public String get(@PathVariable("id") String id) throws Exception {
    …
}
```

Se debe indicar tanto en el RequestMapping a la hora de definir la url a la que responderá la función, como en el parámetro de la función para que lo asocie a éste.

Parque Empresarial PLAZA
C/ Bari 25, Escalera 1 50197 Zaragoza T:+34 902 87 73 92/ F: 976 10 66 21

- Parámetros en el cuerpo de la petición:
  Este tipo de parámetros son comunes en formularios que se envía mediante POST o PUT.

```
@RequestMapping(method = RequestMethod.PUT)
public String get(@RequestBody Map<String, String> params) throws Exception {
    …
}
```

6.3. Usando vistas

Por lo general las vistas se definen en ficheros *.jsp* ubicados en el directorio WEB-INF. Si recordamos la configuración del DispatcherServlet, en él se define un ViewResolver donde se le facilita esta información.

```
@Bean
public ViewResolver viewResolver() {

    InternalResourceViewResolver viewResolver = new InternalResourceViewResolver();
    viewResolver.setViewClass(JstlView.class);
    viewResolver.setPrefix("/WEB-INF/views/");
    viewResolver.setSuffix(".jsp");

    return viewResolver;
}
```

6.4. Tags

6.5. Devolver un JSON

6.6. Recomendaciones REST

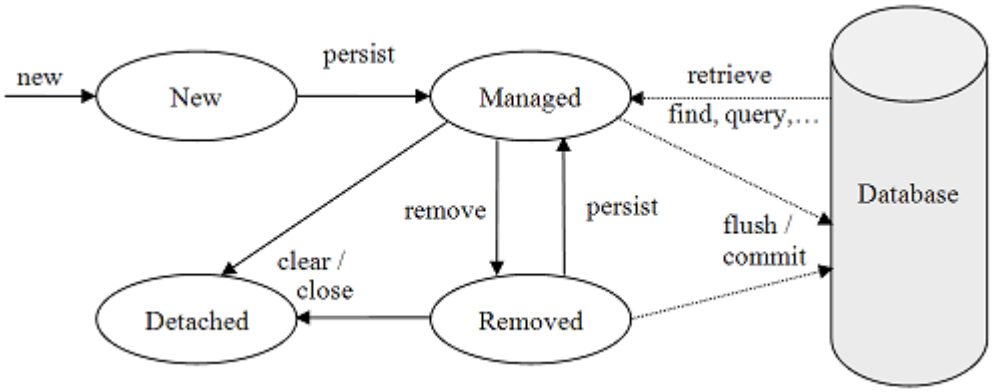
Parque Empresarial PLAZA
C/ Bari 25, Escalera 1 50197 Zaragoza T:+34 902 87 73 92/ F: 976 10 66 21

7. Transaccionalidad

7.1. Ciclo de vida de los objetos en JPA

Cada instancia de un objeto manejado por JPA, tiene un ciclo de vida concreto que hay que entender para ser conscientes de lo que ocurre en el interior del contexto de persistencia en JPA.

Entender y manejar correctamente este ciclo de vida permite no realizar operaciones innecesarias, evitar errores y corregir con mayor celeridad los que aparezcan.



Existen cuatro tipos de estados para estos objetos:

- **New** → Son instancias de objetos creados fuera del persistence context, por tanto, este no es consciente de su existencia.
- **Managed** → Son instancias de objetos marcadas para que el persistence context sea consciente de su existencia, objetos que van a ser persistidos, mergeados, obtenidos de base de datos, etc...
- **Detached** → Son objetos que han dejado de formar parte del persistence context por alguna razón, han sido limpiados, se ha cerrado la transacción, etc...
- **Removed** → Son objetos marcados para ser eliminados de la base de datos.

7.2. SessionFactory y EntityManager

SessionFactory (Hibernate) y EntityManager (JPA) son implementaciones para gestionar sesiones y transacciones sobre bases de datos.

Spring, posee abstracciones para poder trabajar con cualquiera de las dos implementaciones. Entity- Manager ofrece incluso la posibilidad de obtener una Session de Hibernate con la que poder trabajar nativa-mente con todas sus APIs, de hecho, EntityManger (JPA) es posterior a Hibernate, su implementación tomo como referencia el ORM de Jboss, y puede asumirse como una estandarización de este.

Parque Empresarial PLAZA
C/ Bari 25, Escalera 1 50197 Zaragoza T:+34 902 87 73 92/ F: 976 10 66 21

Ambas implementaciones tienen ventajas y desventajas la una frente a la otra, si bien es cierto, como se ha dicho en el párrafo anterior, que desde EntityManager se puede obtener una Session de Hibernate.

No obstante, para la parte de Hibernate de este curso, vamos a trabajar con su implementación nativa, SessionFactory, que ademas, suele ser el estándar hoy por hoy en la mayoría de aplicaciones con Spring.

Configurando un entorno transaccional con SessionFactory

7.3. Gestión de la transaccionalidad

Vas al banco para realizar dos pequeños depósitos y una pequeña retirada:

(Queue)	Permaneces en la fila hasta que el cajero te atienda.
(Open Session)	El cajero te atiende.
(First Transaction)	Realizas el primer depósito.
(Keeping on Transaction)	Realizas el segundo depósito.
(Keeping on/or second Transaction)	Realizas la retirada.
(Close Session)	Te vas.

En este ejemplo de la vida real podemos diferenciar entre una Session (conexión con la base de datos) y una transacción (operación sobre esa conexión con la base de datos).

Spring ofrece una potente API para lidiar con sesiones y ámbitos transaccionales, permitiendo gestionar operaciones de escritura, lectura y rollbacks de forma eficiente y completamente transparentes para el desarrollador.

Como hemos visto anteriormente en el curso, Spring utiliza una combinación de AOP y java proxies para realizar estas técnicas de abstracción.

- El TransactionManager de Spring abre una nueva Session obtenida desde el SessionFactory al invocarse un método anotado con @Transactional.
- Se realizan las operaciones dentro del entorno transaccional.
- El TransactionManager de Spring cierra la transacción y la Session abiertas.

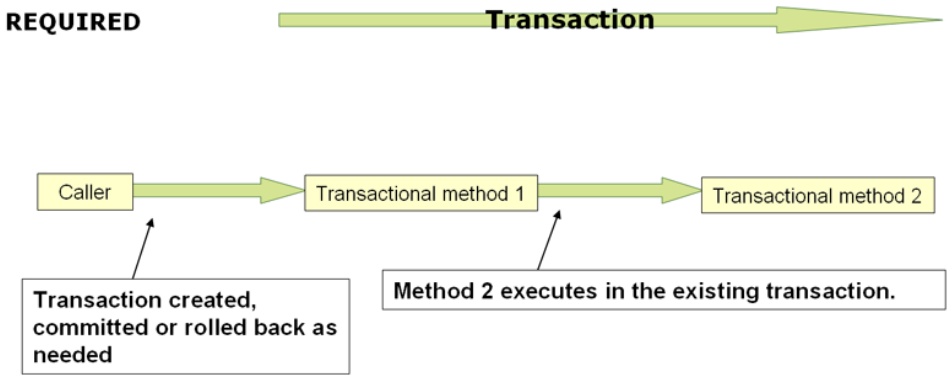
Este comportamiento estándar, es mucho más versátil en realidad, siendo configurable y modificable utilizando diferentes estrategias.

Parque Empresarial PLAZA
C/ Bari 25, Escalera 1 50197 Zaragoza T:+34 902 87 73 92/ F: 976 10 66 21

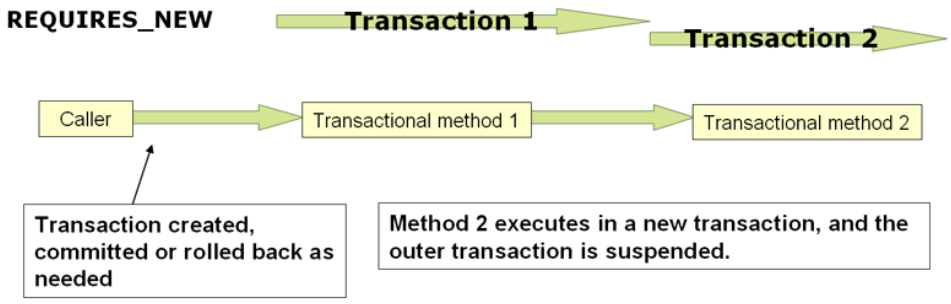
Propagación (Propagation)

Spring permite establecer diferentes estrategias para gestionar el ámbito transaccional en nuestras operaciones sobre la base de datos, existen diferentes estrategias de propagación. Por defecto, el TransactionManager de Spring se configura con Propagation.**REQUIRED**.

- Propagation.**REQUIRED**: Reutiliza una transacción si ya existe, y si no, la crea.



- Propagation.**REQUIRES_NEW**: Crea una nueva transacción en cada método.



- Propagation.**NESTED**: Realiza las operaciones sobre una misma transacción, creando savepoints sobre los que realizar rollbacks.
- Propagation.**MANDATORY:** Asegura que exista transacción en curso, lanzando una excepción en caso contrario.
- Propagation.**NEVER**: Asegura que no exista ninguna transacción en curso, lanzando una excepción en caso contrario.
- Propagation.**NOT_SUPPORTED**: Ejecuta el contenido fuera del método, fuera del entorno transaccional, pausando la transacción en curso.
- Propagation.**SUPPORTS**: Utiliza una transacción que ya existe, y si no se ejecuta en un entorno no transaccional.

Parque Empresarial PLAZA
C/ Bari 25, Escalera 1 50197 Zaragoza T:+34 902 87 73 92/ F: 976 10 66 21

Aislamiento (Isolation)

Spring permite establecer diferentes grados de aislamiento de unas transacciones con otras, de esta manera, se puede permitir a una transacción conocer los cambios que van a ser realizados por transacciones concurrentes, siendo conscientes de los últimos cambios en la base de datos. Por defecto el TransactionManager es configurado con Isolation.**DEFAULT**.

- Isolation.**DEFAULT**: Utiliza el nivel de aislamiento propio del DBMS.
- Isolation.**READ_UNCOMMITED**: Este nivel permite que desde una transacción se puedan leer datos que aún no han sido commiteados en otras transacciones (dirty reads). Pueden suceder además lecturas fantasma (phantom reads, si se hace rollback de la otra transacción y los datos leídos no fueron válidos) y lecturas no repetibles (non-repeatable, una misma consulta puede retornar valores diferentes dentro de la misma transacción).
- Isolation.**READ_COMMITED**: Este nivel no expone datos aún no commiteados por otras transacciones, pero permite que otras transacciones cambien datos recuperados por nuestra transacción. De esta forma, las lecturas sucias se evitan, pero pueden suceder las lecturas fantasma y las lecturas no repetibles.
- Isolation.**REPEATABLE_READ**: Este nivel no permite leer datos aún no commiteados, y tampoco permite modificar los datos leídos por una transacción en curso. Sin embargo, si hacemos 2 consultas (SELECTs) con una condición de rango en el WHERE, y otra transacción inserta datos entre las dos consultas que cumplan con la condición de rango, las consultas retornarán resultados diferentes. Las lecturas sucias y las lecturas no repetibles se evitan, pero pueden suceder las lecturas fantasma.
- Isolation.**SERIALIZABLE**: Además de las condiciones de REPEATABLE_READ, evita el caso de que en que una transacción haga una consulta con una condición de rango, otra transacción inserte datos que cumplan con la condición, y repita la consulta. Ambas consultas retornarán el mismo resultado. Se evitan las lecturas sucias, las lecturas no repetibles y las lecturas fantasma.

RollbackFor y NoRollbackFor

Los atributos rollbackFor y noRollbackFor sirven para especificar bajo qué tipo de excepciones se va lanzar un rollback en esa transacción. Los parámetros permiten arrays con diferentes tipos de excepciones. En el caso de noRollbackFor sirve para especificar casos concretos de excepciones sobre las que no se lanzará.

Parque Empresarial PLAZA
C/ Bari 25, Escalera 1 50197 Zaragoza T:+34 902 87 73 92/ F: 976 10 66 21

7.4. Repositorio genérrico

Parque Empresarial PLAZA
C/ Bari 25, Escalera 1 50197 Zaragoza T:+34 902 87 73 92/ F: 976 10 66 21

8. Spring Security

Spring Security ofrece una solución integral de seguridad del control de la autenticación y autorización tanto a nivel de peticiones web como a nivel de invocación de clases y métodos.

La capa de seguridad establece una cadena de filtros (FilterChain) por la que pasan las peticiones que llegan al servidor.

Cada uno de los filtros de esta cadena tiene una ordenación en la cadena y una funcionalidad específica, las cuales deben ser respetadas para garantizar el correcto funcionamiento de Spring Security.

Además, Spring Security permite añadir, eliminar y modificar filtros de la FilterChain con el fin de adecuar el marco de seguridad a las características concretas del proyecto actual.

Entre las principales competencias de Spring Security podemos encontrar:

- Control de peticiones al servidor e invocación de clases y métodos.
- Autenticación de usuarios / administración de roles.
- Control de sesiones.
- Control de cookies de autenticación.

8.1. FilterChain

Es el núcleo principal de Spring Security, por ella pasan todas las peticiones realizadas al servidor, para ello, se registra un Servlet Filter proveído por Spring, **DelegatingFilterProxy**, registrado mediante la clase **AbstractSecurityWebApplicationInitializer** (o en el web.xml de la aplicación), dicho filtro debe configurarse en las primeras posiciones (excepto filtros de logs) para garantizar la seguridad en la mayor parte de los componentes de la aplicación.

Cada petición recorre la FilterChain de forma secuencial, delegando en cada uno de los filtros que la componen una funcionalidad especifica. Resulta fundamental respetar el orden de los filtros, ya que cada uno puede resolver o resuelven dependencias que afectan al comportamiento de los filtros posteriores.

El orden predefinido de la FilterChain es el siguiente:

- **ChannelProcessingFilter** → Redirecciona a un protocolo diferente.
- **SecurityContextPersistenceFilter** → Realiza una copia del SecurityContext para poder ser guardada en la HttpSession y ser reutilizada en peticiones posteriores.
- **ConcurrentSessionFilter** → Actualiza la SessionRegistry para que se vean reflejados los cambios que se han podido producir por el SecurityContextPersistenceFilter.
- **UsernamePasswordAuthenticationFilter, BasicAuthenticationFilter, CasAuthenticationFilter,...** → Mecanismos de autentificación.

Parque Empresarial PLAZA
C/ Bari 25, Escalera 1 50197 Zaragoza T:+34 902 87 73 92/ F: 976 10 66 21

- **SecurityContextHolderAwareRequestFilter** → Encapsula la ServletRequest con métodos de seguridad.
- **JaasApiIntegrationFilter** → Procesa la FilterChain para autentificaciones JAAS.
- **RememberMeAuthenticationFilter** → Si la petición dispone de una cookie de sesión, la procesa.
- **AnonymousAuthenticationFilter** → Si los métodos de autentificación anteriores fallan, añade una Anony-mous Authentication en el SecurityContext.
- **ExceptionTranslationFilter** → Procesa las excepciones para devolver un código http.
- **FilterSecurityInterceptor** → Protege URLs y excepciones cuando el acceso es denegado.

8.2. Tipos de componentes

Spring Security desarrolla su funcionalidad mediante una serie de componentes con unas responsabilidades determinadas.

- **Handlers →** Son los encargados de determinar la redirección del usuario en determinadas situaciones. Son inyectados en los Filters, y lanzados a la hora de tener que tomar una decisión, generalmente para resolver la respuesta.

- **Providers →** Son los encargados en última instancia de realizar la lógica necesaria para resolver una autentificación contra cualquier tipo de repositorio de datos y crear el objeto Authentication. Son inyectados en los AuthenticationManagers, y lanzados después realizar las verificaciones necesarias antes de comenzar el proceso de autentificación.

- **AuthenticationManager →** Disponen de una cadena de Providers para realizar difefentes estrategias de autentificación, generalmente, habrá un Provider por AuthenticationManager, pero si es necesario, puede haber más. Al igual que los handlers, son inyectados en los Filters, y lanzados a la hora de realizar el proceso de autentificación después de haberse realizado las validaciones previas del Filter.

- **Filters →** Son los componentes básicos de la FilterChain, en ellos, se desarrollan las estrategias y lógica a realizar desde la FilterChain, gestionan Handlers y AuthenticationManagers. Son inyectados en la FilterChain.

8.3. Configuración

Parque Empresarial PLAZA
C/ Bari 25, Escalera 1 50197 Zaragoza T:+34 902 87 73 92/ F: 976 10 66 21

8.4. Taglib

Spring Security proporciona un Taglib que nos da acceso en la JSP a las propiedades del usuario en SecurityContext, facilitando la construcción de la vista con las opciones de seguridad.

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

Authentication Tag: Este tag da acceso al objeto Authentication de SecurityContext, mediante el atributo "property" nos permite acceder a una serie de propiedades de este objeto.
La interface Authentication expone los siguientes métodos:

Método	Descripción
authorities	Colección de roles (GrantedAuthority).
credentials	Contraseña
details	Detalles del proceso de autentificación. NULL por defecto.
principal	Información del usuario autenticado. Por defecto una implementación de UserDetails. En IDIA el objeto ClientAuth.
isAuthenticated	Devuelve true o false dependiendo del estado de autentificación del usuario.

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

Además podemos utilizar los siguientes atributos opcionales:

Atributo	Descripción
var	Variable de página que guarda el objeto devuelto de accedido desde "property".
scope	Ámbito de "var". (page, request, session o application).
htmlEscape	true o false. Habilita el escape HTML del resultado de "property".

Authorization Tag: Este tag nos permite evaluar los permisos del usuario según su colección de roles. Mediante el atributo "access" podemos definir las condiciones de acceso según estos roles. Para poder definir estas condiciones podemos utilizar SpEL.

Parque Empresarial PLAZA
C/ Bari 25, Escalera 1 50197 Zaragoza T:+34 902 87 73 92/ F: 976 10 66 21

Lista de expresiones SpEL:

Expresión	Descripción
hasRole("ROLE")	true si el usuario actual tiene ese rol.
hasAnyRole(["ROLE_1", "ROLE_2"])	true si el usuario actual tiene alguno de los roles.
principal	Acceso a las propiedades de principal.
authentication	Acceso a las propiedades de authentication.
permitAll	Siempre true.
denyAll	Siempre false.
isAnonymous()	true si el usuario no está autenticado.
isRememberMe()	true si el usuario esta autenticado mediante el Remember.
isAuthenticated()	true si el usuario esta autenticado.
isFullyAuthenticated()	true si el usuario no es anónimo ni recordado.

```
<sec:authorize access="hasRole(ROLE)">
    …
</sec:authorize>
```

Además podemos utilizar los siguientes atributos opcionales:

Atributo	Descripción
url	True si el usuario tiene acceso a la url relativa especificada.
method	Se combina con "url", sirve para establecer el método HTTP concreto de acceso.
var	Guarda una variable de página con el resultado de la operación.

```
<sec:authorize access="hasRole(ROLE)" var="isRole">
    …
</sec:authorize>

<c:if test="${isRole}">
    …
</c:if>
```


Parque Empresarial PLAZA
C/ Bari 25, Escalera 1 50197 Zaragoza T:+34 902 87 73 92/ F: 976 10 66 21

9. Spring Test

Se trata de un módulo para realizar test unitarios sobre los proyectos realizados con Spring. Su principal característica, consiste en crear un Application Context a partir de las clases de configuración o xml de configuración de Spring. Al igual que el Application Context de la aplicación, se resuelven las dependencias mediante IoC y se siguen los mismos mecanismos de inicialización de beans, creando un Application Context espejo listo para ser utilizado desde los test unitarios.

Aunque se pueden crear pequeñas configuraciones del Application Context que se centren en levantar únicamente los componentes que van a intervenir en unos determinados test unitarios, normalmente, la velocidad de despliegue del ApplicationContext acaba siendo en gran medida ocasionada por componentes concretos, como la inicialización de conexiones a base de datos, etc... Siendo estos métodos, de uso transversal a cualquier servicio o método de servicio de la aplicación. Por esta razón, es buena idea la implementación de una clase central abstracta, que guarde la configuración mínima necesaria para crear un Application Context, y que sea extendida por parte de las clases específicas que implementen los test unitarios.

```
@RunWith(SpringJUnit4ClassRunner.class)
@ActiveProfiles("test")
@Rollback
@WebAppConfiguration
@ContextConfiguration(classes = {ApplicationConfig.class, RestSecurityConfig.class,
DispatcherServletConfig.class})
@TestExecutionListeners(
        listeners = {
                ServletTestExecutionListener.class,
                DependencyInjectionTestExecutionListener.class,
                DirtiesContextTestExecutionListener.class,
                TransactionalTestExecutionListener.class,
                WithSecurityContextTestExecutionListener.class})
public abstract class AppContextConfigurationAware {
    @Autowired
    protected WebApplicationContext webApplicationContext;

    @Autowired
    @SuppressWarnings("SpringJavaAutowiringInspection")
    private FilterChainProxy springSecurityFilterChain;

    protected MockMvc mockMvc;

    @Before
    public void setup() {

        this.mockMvc = MockMvcBuilders
                .webAppContextSetup(this.webApplicationContext)
                .apply(SecurityMockMvcConfigurers.springSecurity(this.springSecurityFilterChain))
                .build();
    }

}
```


Parque Empresarial PLAZA
C/ Bari 25, Escalera 1 50197 Zaragoza T:+34 902 87 73 92/ F: 976 10 66 21

- **@RunWith (con SpringJUnit4ClassRunner.class)** → Ejecuta el entorno de pruebas mediante el Runner proveído por Spring.
- **@ActiveProfiles** → Especifica, si se está usando @Profile en los beans, que perfiles están activos en los test unitarios.
- **@Rollback** → Especifica que cualquier transacción acabara ejecutando un rollback sobre la base de datos.
- **@WebAppConfiguration** → Indica a Spring que debe crear un WebApplicationContext para los test unitarios.
- **@ContextConfiguration** → Especifica las clases de configuración del Application Context.
- **@TestExecutionListeners** → Define una API de Listeners que reaccionan a los eventos generados por el TestContextManager.

Spring provee una potente herramienta, MockMvc, para interaccionar directamente con peticiones Http.

```
protected MockMvc mockMvc;
```

Para resolver sus dependencias, se necesita obtener las instancias del **ApplicationContext** creado por el **ContextLoaderListener** y de la **FilterChainProxy** creada por el **DelegationFilterProxy**.

```
@Autowired
protected WebApplicationContext webApplicationContext;

@Autowired
private FilterChainProxy springSecurityFilterChain;
```

Mediante la anotación @Before, se inicializa la instancia de MockMvc para ser utilizada desde los métodos que desarrollan las funcionalidades de los test unitarios.

Parque Empresarial PLAZA
C/ Bari 25, Escalera 1 50197 Zaragoza T:+34 902 87 73 92/ F: 976 10 66 21

10. Hibernate

10.1. Fundamentos de Hibernate

10.2. Hibernate Session

Desde Hibernate, la interface Session nos provee diferentes métodos para manejar el ciclo de vida de estos objetos.

- **Contains** → Nos permite conocer si un objeto ya está siendo manejado por el persistence context.
- **Delete** → Marca un objeto para ser eliminado al finalizar la transacción.
- **Evict** → Marca un objeto para dejar de ser manejado por el persistence context.
- **Merge** → Crea una copia de un objeto, y la marca para ser manejada por el persistence context, la copia original (de la que tenemos referencia) queda fuera del persistence context, por lo que los cambios posteriores no se verán reflejados en la base de datos.
- **Persist** → Marca un objeto para ser manejado por el persistence context y persistido en base de datos al finalizar la transacción.
- **Refresh** → Actualiza el estado de un objeto desde el persistence context.
- **Update** → Marca un objeto detached para ser manejado por el persistence context y actualizado al finalizar la transacción.

Para que un objeto sea managed, debemos estar en ámbito transaccional, sino, dicho objeto nunca podría estar en este estado.

Cuando un objeto está marcado como managed, es decir, el persistence context es consciente y está manejando este objeto, cualquier cambio realizado sobre este objeto, o cualquiera de sus Collections dependiendo de su CascadeType marcan automáticamente el objeto como update, es decir, no es necesario especificar mediante el método update() de la interface Session que ese objeto debe ser actualizado.

10.3. Entidades

Las entidades son las clases utilizadas para mapear una tabla de la base de datos al modelo de objetos de Hibernate.

Parque Empresarial PLAZA
C/ Bari 25, Escalera 1 50197 Zaragoza T:+34 902 87 73 92/ F: 976 10 66 21

```java
@Entity(name = "vehicle")
@Table(name = "vehicle")
public class Vehicle {

    @Id
    @GeneratedValue(strategy = GenerationType.AUTO)
    @Column(name = "id")
    private int id;

    @Column(name = "name")
    private String name;
    …
}
```

- @Entity:
  - name: Define el nombre de la entidad.
- @Table:
  - name: Define el nombre de la tabla.
- @Id: Indica que ese atributo es *PRIMARY KEY*.
- @GeneratedValue: Defina la forma en la que se van a generar los identificadores. *GenerationType.AUTO* toma la estrategia del motor de base de datos.

Propiedades

Todas las propiedades no estáticas y no marcadas como *transient* son consideradas propiedades que deben ser persistidas.

```java
@Transient
private int transientProperty; // non-persistent property

private static String staticProperty; // non-persistent property

@Column(name = "name")
private String name; // persistent property
```

Las propiedades que deben ser persistidas han de ser mapeadas con sus correspondientes columnas de la base de datos, esto se define con **@Column**. Sus principales propiedades son:

- **name**: Nombre de la columna de la base de datos.
- **unique**: Si debe tener un valor único.
- **nullable**: Si puede tomar valor null.
- **insertable**: Si se debe insertar en la base de datos.
- **updatable**: Si se debe actualizar su valor en la base de datos.
- **length**: Longitud del campo.

Parque Empresarial PLAZA
C/ Bari 25, Escalera 1 50197 Zaragoza T:+34 902 87 73 92/ F: 976 10 66 21

10.4. Colecciones

Usando anotaciones se pueden mapear relaciones en Collection, List, Map y Set usando @OneToMany o @ManyToMany, y sobre una entidad utilizando @ManyToOne o @OneToOne. Estas relaciones pueden establecerse de forma unidireccional o de forma bidireccional.

- **Unidireccionales:** Las relaciones unidireccionales se definen sobre una propiedad de una entidad. Al ser unidireccional, solo la entidad que posee la definición tiene acceso a esta relación. Un ejemplo simple de una relación unidireccional podría ser el siguiente:

```
@Entity(name = "vehicle")
@Table(name = "vehicle")
public class Vehicle {

    @Id
    @GeneratedValue(strategy = GenerationType.AUTO)
    @Column(name = "id")
    private int id;

    …
    @ManyToOne
    @JoinColumn(name = "vehicle_type_id", referencedColumnName = "id")
    private VehicleType vehicleType;
    …
}

@Entity(name = "vehicleType")
@Table(name = "vehicle_type")
public class VehicleType {

    @Id
    @GeneratedValue(strategy = GenerationType.AUTO)
    @Column(name = "id")
    private int id;
    …
}
```

En este ejemplo se define una relación N-1 sobre la propiedad *vehicleType*, varios vehículos pueden pertenecer al mismo tipo. Desde la entidad *Vehicle* es posible conocer el tipo, sin embargo desde la entidad *VehicleType* la relación no es visible.

- **Bidireccionales:** Las relaciones bidireccionales se definen sobre la propiedad de una entidad, y en la otra entidad afectada se indica cual es la propiedad que posee la definición. De esta forma ambas entidades tienen acceso a la relación. Por ejemplo:

Parque Empresarial PLAZA
C/ Bari 25, Escalera 1 50197 Zaragoza T:+34 902 87 73 92/ F: 976 10 66 21

```
@Entity(name = "sale")
@Table(name = "sale")
public class Sale {



    @Id
    @GeneratedValue(strategy = GenerationType.AUTO)
    @Column(name = "id")
    private int id;
    …
    @ManyToOne
    @JoinColumn(name="vehicle_id")
    private Vehicle vehicle;


    …
}

@Entity(name = "vehicle")
@Table(name = "vehicle")
public class Vehicle {
    @Id
    @GeneratedValue(strategy = GenerationType.AUTO)
    @Column(name = "id")
    private int id;
    …
    @OneToMany(fetch = FetchType.LAZY, mappedBy = "vehicle")
    private List<Sale> sales;
    …
}
```

En el ejemplo anterior, la entidad *Sale* define una relación ManyToOne sobre la propiedad *vehicle*, y la entidad *Vehicle* indica que tiene una relación OneToMany y que la definición de ésta se encuentra en la propiedad *vehicle* (de la entidad *Sale*). De esta forma, ambas entidades tienen acceso a la relación.

Tipos de relaciones

- **@OneToOne:** Define una asociación de un solo valor a otra entidad con una relación 1-1.

```
// In CustomerSocialNetworks:
@OneToOne
@JoinColumn(name = "customer_id")
private Customer customer;
```


Parque Empresarial PLAZA
C/ Bari 25, Escalera 1 50197 Zaragoza T:+34 902 87 73 92/ F: 976 10 66 21

```
// In Customer:
@OneToOne(mappedBy = "customer")
private CustomerSocialNetworks customerSocialNetworks;
```

- **@OneToMany:** Define una relación con multiplicidad 1 – N.
  - Unidireccional:

```
// In Customer:
@OneToMany(fetch = FetchType.EAGER)
@JoinColumn(name = "customer_id")
private Set<CustomerAddress> customerAddresses;
```

  - Bidireccional:

```
// In Customer:
@OneToMany(mappedBy = "customer")
private Set<Sale> sales;

// In Sale:
@ManyToOne
@JoinColumn(name="customer_id")
private Customer customer;
```

- **@ManyToOne:** Define un solo valor que se relaciona con la entidad con multiplicidad N – 1.

```
// In Vehicle
@ManyToOne
@JoinColumn(name = "vehicle_type_id", referencedColumnName = "id")
private VehicleType vehicleType;
```

- **@ManyToMany:** Define una colección de valores correspondientes a una relación de multiplicidad N – N con otra entidad.

```
// In Customer:
@ManyToMany
@Fetch(FetchMode.SELECT)
@JoinTable(name = "customer_favorites_vehicle_types",
joinColumns = @JoinColumn(name = "customer_id", referencedColumnName = "id"),
inverseJoinColumns = @JoinColumn(name = "vehicle_type_id", referencedColumnName = "id"))
private Set<VehicleType> favoritesVehicleTypes;
```


Parque Empresarial PLAZA
C/ Bari 25, Escalera 1 50197 Zaragoza T:+34 902 87 73 92/ F: 976 10 66 21

```
// In VehicleType
@ManyToMany(mappedBy = "favoritesVehicleTypes")
private Set<Customer> customers;
```

Los parámetros más importantes que pueden recibir las anotaciones anteriores son:

- **fetch**: Esta propiedad puede tomar dos valores:
  - **FetchType.EAGER**: Indica que los datos serán recuperados junto a la entidad que define la relación.
  - **FetchType.LAZY**: Indica que los datos no se recuperarán en el momento de obtener la entidad que define la relación. Las relaciones con esta propiedad se recuperan la primera vez que sean accedidos dentro del contexto transaccional (si se encuentra fuera provocará una excepción).

  Por defecto esta propiedad toma el valor FetchType.EAGER en las relaciones que definen un solo valor (@OneToOne, @ManyToOne), y FetchType.LAZY en las que definen una colección (@OneToMany, @ManyToMany).

- **mappedBy** (salvo @ManyToOne): En relaciones bidireccionales, indica el nombre de la propiedad de la otra entidad afectada, en la cual se define la relación.

Parque Empresarial PLAZA
C/ Bari 25, Escalera 1 50197 Zaragoza T:+34 902 87 73 92/ F: 976 10 66 21