Spring MVC

Contenidos

- Spring con Spring Boot
- IoC con Spring Core
- Acceso a datos con Spring Data
- Spring MVC
  - Controladores
  - Vistas
- Seguridad
- Servicios Rest con Spring
- Clientes de los Servicios Rest

Enlaces

- Spring:
  - https://spring.io/projects
- Spring Core
  - https://docs.spring.io/spring-framework/docs/current/spring-framework-reference/core.htm
- Spring Data
  - https://docs.spring.io/spring-data/jpa/docs/2.1.5.RELEASE/reference/html/
- Spring MVC
  - https://docs.spring.io/spring/docs/current/spring-framework-reference/web.html
- Spring HATEOAS
  - https://docs.spring.io/spring-hateoas/docs/0.25.1.RELEASE/reference/html
- Spring Data REST
  - https://docs.spring.io/spring-data/rest/docs/3.1.5.RELEASE/reference/html/
- Ejemplos:
  - https://github.com/spring-projects/spring-data-examples
  - https://github.com/spring-projects/spring-data-rest-webmvc
  - https://github.com/spring-projects/spring-hateoas-examples
  - https://github.com/spring-projects/spring-integration-samples

EVOLUCIÓN DEL DESARROLLO WEB

1990

Cliente

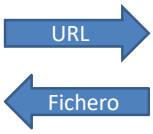
Tim Berners-Lee (CERN)

- ENQUIRE
  - Hipertexto
  - HTTP
  - HTML

Servidor

Robert Cailliau

- CERN httpd
  - Servidor de ficheros
  - HTTP – URL
  - Ficheros estáticos de texto



Origen

- La Web no fue concebida para el desarrollo de aplicaciones. El problema que se pretendía resolver su inventor, Tim Berners-Lee, era el cómo organizar información a través de enlaces.
- De hecho la Web nació en el laboratorio de partículas CERN básicamente para agrupar un conjunto muy grande de información y datos del acelerador de partículas que se encontraba muy dispersa y aislada.
- Mediante un protocolo muy simple (HTTP), un sistema de localización de recursos (URL) y un lenguaje de marcas (HTML) se podía poner a disposición de todo científico en el mundo la información existente en el CERN de tal forma que mediante enlaces se pudiese acceder a información relacionada con la consultada.

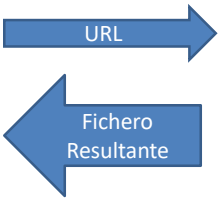
HTML

Cliente

- Navegadores
  - HTML

Servidor

- Servidor Web
  - Servidor de ficheros
  - Ficheros estáticos
    - Texto
    - HTML
  - Ficheros dinámicos
    - CGI (C, Perl)
    - …
    - Servlet



CGI

- Por la necesidad que el servidor Web pudiese devolver páginas Web dinámicas y no únicamente contenido estático residente en ficheros HTML se desarrolló la tecnología CGI (Common Gateway Interface) donde el servidor Web invocaba un programa el cual se ejecutaba, devolvía la página Web y el servidor Web remitía este flujo de datos al navegador.
- Un programa CGI podía ser cualquier programa que la máquina pudiese ejecutar: un programa en C, o en Visual Basic o en Perl. Normalmente se elegía este último por ser un lenguaje de script el cual podía ser traslado con facilidad de una arquitectura a otra. CGI era únicamente una pasarela que comunicaba el servidor Web con el ejecutable que devolvía la página Web.

Formularios

Cliente

- Navegadores
  - HTML 3.2
  - Formularios
  - Plug-in
    - Applet
    - ActiveX
  - Scripting de cliente
    - JavaScript

Servidor

- Servidor Web
  - Servidor de ficheros
  - Ficheros estáticos
    - Texto, HTML, Imágenes, …
  - Ficheros dinámicos
  - Sricpting de servidor
    - ASP
    - PHP
    - …
    - JSP

Scripting

- CGI era una solución cómoda de realizar páginas Web dinámicas pero tenía un grave problema de rendimiento que lo hizo insostenible en cuanto la demanda de la Web comenzó a disparar las peticiones de los servidores Web.
- Para agilizar esto, los principales servidores Web del momento (Netscape e IIS) desarrollaron un sistema para la ejecución dinámica de aplicaciones usando el propio contexto del servidor Web. En el caso de Netscape se le denominó NSAPI (Netscape Server Application Program Interface) y en el caso de IIS se le llamó ISAPI.

Web 2.0

Cliente

- Navegadores
  - HTML 4
  - CSS
  - DOM
  - AJAX
  - RIA
    - Shockwave Flash
    - Silverlight
  - JS Framework

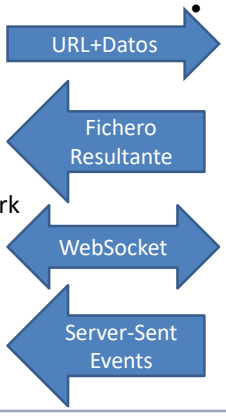
Servidor

- Servidor Web
  - Servidor de ficheros
  - Ficheros estáticos
  - Ficheros dinámicos
  - Sricpting de servidor
  - Servidor de aplicaciones
    - ASP.NET
    - J2EE
  - Web Services
    - WS XML
    - RestFul

Actualidad

Cliente

- Navegadores
  - HTML 5
  - CSS 3
  - ~~RIA~~
  - Móviles
  - JS RIA Framework
  - EcmaScript 6



Servidor

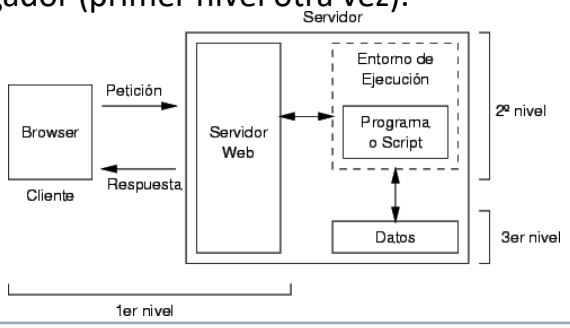
- Servidor Web
  - Servidor de ficheros
  - Ficheros estáticos
  - Ficheros dinámicos
  - Sricpting de servidor
  - Servidor de aplicaciones
    - NodeJS
  - Web Services
  - Server-Sent Events
    - Notificaciones PUSH

Aplicación Web

- Conjunto de páginas que residen en un directorio web y sus subdirectorios.
- Cualquier página puede ser el punto de entrada de la aplicación, no se debe confundir con el concepto de página principal con el de página por defecto.
- Externamente, no existe el concepto de aplicación como entidad única.
- Internamente, dependiendo de la tecnología utilizada una aplicación puede ser una entidad única o una colección de objetos independientes.

Arquitectura

- Una aplicación Web típica recogerá datos del usuario (primer nivel), los enviará al servidor, que ejecutará un programa (segundo y tercer nivel) y cuyo resultado será formateado y presentado al usuario en el navegador (primer nivel otra vez).



Ventajas e inconvenientes

Ventajas

- Inmediatez y accesibilidad
- No ocupan espacio local
- Actualizaciones inmediatas
- No hay problemas de compatibilidad
- Multiplataforma
- Consumo de recursos bajo
- Portables
- Alta escalabilidad y disponibilidad

Inconvenientes

- Interfaz de interacción con el usuario muy limitada
- Base tecnológica inadecuada
- Incompatibilidades entre navegadores
- Bajo rendimiento al ser interpretado
- Cesión tecnológica
- Seguridad menos robusta

Single-page application (SPA)

- Un single-page application (SPA), o aplicación de página única es una aplicación web o es un sitio web que utiliza una sola página con el propósito de dar una experiencia más fluida a los usuarios como una aplicación de escritorio.
- En un SPA todo el códigos de HTML, JavaScript y CSS se carga de una sola vez o los recursos necesarios se cargan dinámicamente cuando lo requiera la página y se van agregando, normalmente como respuesta de los acciones del usuario.
- La página no se tiene que cargar otra vez en ningún punto del proceso, tampoco se transfiere a otra página, aunque las tecnologías modernas (como el pushState() API del HTML5) pueden permitir la navegabilidad en páginas lógicas dentro de la aplicación.
- La interacción con las aplicaciones de página única pueden involucrar comunicaciones dinámicas con el servidor web que está por detrás, habitualmente utilizando AJAX o WebSocket (HTML5).

Estado actual de la adopción de los nuevos estándares

- http://caniuse.com/
- HTML 5
  - http://html5test.com
  - http://html5demos.com
- CSS
  - http://css3test.com/
- EcmaScript
  - https://kangax.github.io/compat-table/es6/
- Navegadores mas utilizados
  - http://www.netmarketshare.com/

http://spring.io

SPRING CON SPRING BOOT

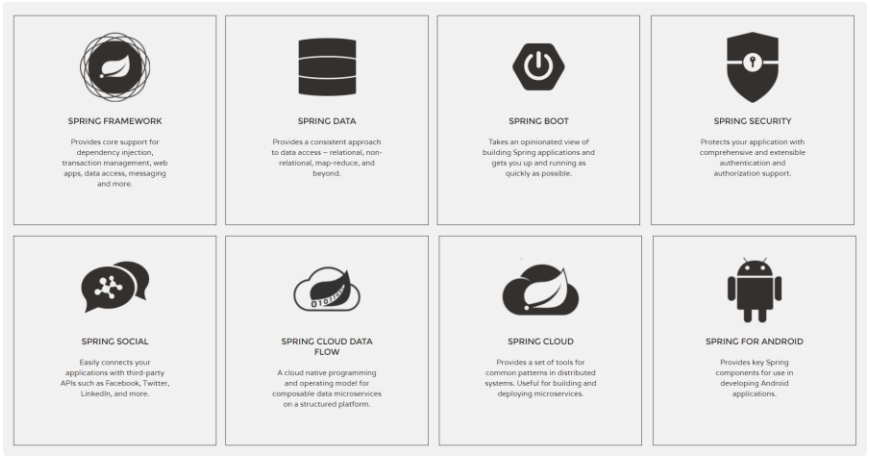
Spring

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

Características

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

Proyectos



Módulos necesarios

- Spring Framework
  - Spring Core
    - Contenedor IoC (inversión de control) - inyector de dependencia
  - Spring MVC
    - Framework basado en MVC para aplicaciones web y servicios REST
- Spring Data
  - Simplifica el acceso a los datos: JPA, bases de datos relacionales / NoSQL, nube
- Spring Boot
  - Simplifica el desarrollo de Spring: inicio rápido con menos codificación

Spring Boot

- Spring tiene una gran cantidad de módulos que implican multitud de configuraciones. Estas configuraciones pueden requerir mucho tiempo, pueden ser desconocidas para principiantes y suelen ser repetitivas.
- La solución de Spring es Spring Boot, que aplica el concepto de Convention over Configuration (CoC).
- CoC es un paradigma de programación que minimiza las decisiones que tienen que tomar los desarrolladores, simplificando tareas.
- No obstante, la flexibilidad no se pierde, ya que a pesar de otorgar valores por defecto, siempre se puede configurar de forma extendida.
- De esta forma se evita la repetición de tareas básicas a la hora de construir un proyecto.
- Spring Boot es una herramienta que nace con la finalidad de simplificar aun más el desarrollo de aplicaciones basadas en el framework Spring Core: que el desarrollador solo se centre en el desarrollo de la solución, olvidándose por completo de la compleja configuración que actualmente tiene Spring Core para poder funcionar.

Spring Boot

- Configuración:
  - Spring Boot cuenta con un complejo módulo que autoconfigura todos los aspectos de nuestra aplicación para poder simplemente ejecutar la aplicación, sin tener que definir absolutamente nada.
- Resolución de dependencias:
  - Con Spring Boot solo hay que determinar que tipo de proyecto estaremos utilizando y el se encarga de resolver todas las librerías/dependencias para que la aplicación funcione.
- Despliegue:
  - Spring Boot se puede ejecutar como una aplicación Stand-alone, pero también es posible ejecutar aplicaciones web, ya que es posible desplegar las aplicaciones mediante un servidor web integrado, como es el caso de Tomcat, Jetty o Undertow.

Spring Boot

- Métricas:
  - Por defecto, Spring Boot cuenta con servicios que permite consultar el estado de salud de la aplicación, permitiendo saber si la aplicación está encendida o apagada, memoria utilizada y disponible, número y detalle de los Bean's creado por la aplicación, controles para el prendido y apagado, etc.
- Extensible:
  - Spring Boot permite la creación de complementos, los cuales ayudan a que la comunidad de Software Libre cree nuevos módulos que faciliten aún más el desarrollo.
- Productividad:
  - Herramientas de productividad para desarrolladores como Spring Initializr, Lombok, LiveReload y Auto Restart, funcionan en su IDE favorito: Spring Tool Suite, IntelliJ IDEA y NetBeans.

Con Eclipse

- Descargar Hibernate:
  - http://hibernate.org/orm/downloads/
- Descargar e instalar JDK:
  - http://www.oracle.com/technetwork/java/javase/downloads/jdk8-downloads-2133151.html
- Descargar y descomprimir Eclipse:
  - https://www.eclipse.org/downloads/
- Añadir a Eclipse las Hibernate Tools
  - Help > Eclipse Marketplace: JBoss Tools
- Crear una User Librarie para Hibernate
  - Window > Preferences > Java > Build Path > User Libraries > New
  - Add External JARs: \lib\required
- Descargar y registrar la definición del driver JDBC
    - Window > Preferences > Data Management > Connectivity > Driver Definition > Add

Instalación Spring Tool Suite

- https://spring.io/tools
- Spring Tool Suite
  - IDE gratuito, personalización del Eclipse
- Plug-in para Eclipse (VSCode, Atom)
  - Help → Eclipse Marketplace …
    - Spring Tools 4 for Spring Boot

devtools

- Al realizar nuevos cambios en nuestra aplicación, podemos hacer que el arranque se reinicie automáticamente. Para eso es necesario incluir una dependencia Maven extra: spring-boot-devtools.
- Durante el tiempo de ejecución, Spring Boot supervisa la carpeta que se encuentra en classpath (en maven, las carpetas que están en la carpeta "target"). Solo necesitamos activar la compilación de las fuentes en los cambios que causarán la actualización de la carpeta 'destino' y Spring Boot reiniciará automáticamente la aplicación. Si estamos utilizando Eclipse IDE, la acción de guardar puede desencadenar la compilación.
- El módulo spring-boot-devtools incluye un servidor LiveReload incorporado que se puede usar para activar una actualización del navegador cuando se cambia un recurso. Las extensiones del navegador LiveReload están disponibles gratuitamente para Chrome, Firefox y Safari desde:
  http://livereload.com/extensions/
  spring.devtools.restart.additional-paths=.
  spring.devtools.livereload.enabled=true

Crear proyecto

- Desde Spring Initializr:
  - https://start.spring.io/
  - Descomprimir en el workspace
  - Import → Maven → Existing Maven Project
- Desde Eclipse:
  - New Project → Sprint Boot → Spring Started Project
- Dependencias
  - Web
  - JPA
  - JDBC (o proyecto personalizado)

pom.xml

```
<parent>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-parent</artifactId>
    <version>2.1.1.RELEASE</version>
</parent>
<dependencies>
    <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-web</artifactId>
    </dependency>
</dependencies>
```

- Con las etiquetas <parent></parent> se indica que nuestro POM hereda del de Spring Boot. La dependencia spring-boot-starter-web es la necesaria para poder empezar con un proyecto de tipo MVC, pero a medida que crece la aplicación se iran añadiendo más dependencias.
- Para alterar la configuración dada por defecto se ofrecen campos que se añadirán y asignaran en el archivo application.properties de la carpeta src/main/resources del proyecto.

Application

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

@SpringBootApplication
public class DemoApplication implements CommandLineRunner {

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

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

    }

}
```


Configuración

- @Configuration: Indica que esta es una clase usada para configurar el contenedor Spring.
- @ComponentScan: Escanea los paquetes de nuestro proyecto en busca de los componentes que hayamos creado, ellos son, las clases que utilizan las siguientes anotaciones: @Component, @Service, @Controller, @Repository.
- @EnableAutoConfiguration: Habilita la configuración automática, esta herramienta analiza el classpath y el archivo application.properties para configurar nuestra aplicación en base a las librerías y valores de configuración encontrados, por ejemplo: al encontrar el motor de bases de datos H2 la aplicación se configura para utilizar este motor de datos, al encontrar Thymeleaf se crearan los beans necesarios para utilizar este motor de plantillas para generar las vistas de nuestra aplicación web.
- @SpringBootApplication: Es el equivalente a utilizar las anotaciones: @Configuration, @EnableAutoConfiguration y @ComponentScan

Configuración

- Editar src/main/resources/application.properties:

```
# Oracle settings
spring.datasource.url=jdbc:oracle:thin:@localhost:1521:xe
spring.datasource.username=hr
spring.datasource.password=hr
spring.datasource.driver-class-name=oracle.jdbc.driver.OracleDriver

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

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

server.port=8080
```

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

Oracle Driver con Maven

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

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

Configuración del proxy: Maven

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

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

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

Instalación de MySQL

- Descargar e instalar:
  - https://mariadb.org/download/
- Incluir en la seccion [mysqld] de %MYSQL_ROOT%/data/my.ini
  - default_time_zone='+01:00'
- Descargar bases de datos de ejemplos:
  - https://dev.mysql.com/doc/index-other.html
- Instalar bases de datos de ejemplos:
  - mysql -u root -p < employees.sql
  - mysql -u root -p < sakila-schema.sql
  - mysql -u root -p < sakila-data.sql

JSP

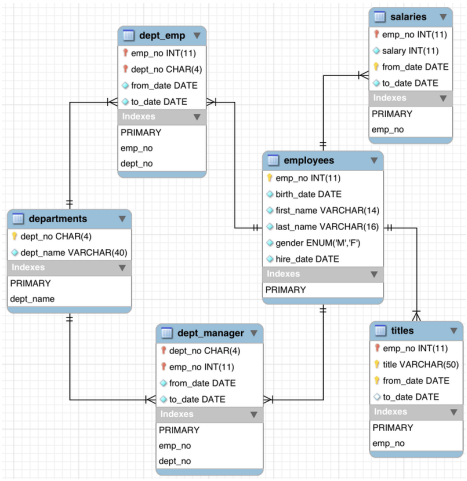
- En el fichero pom.xml:

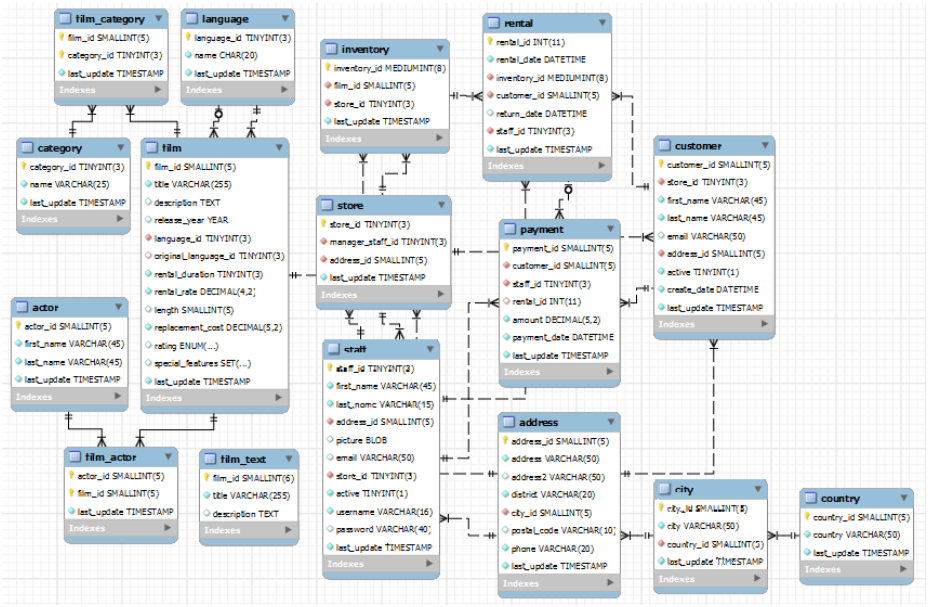
```
<dependency>
        <groupId>org.apache.tomcat.embed</groupId>
        <artifactId>tomcat-embed-jasper</artifactId>
        <scope>provided</scope>
 </dependency>
<dependency>
       <groupId>javax.servlet</groupId>
       <artifactId>jstl</artifactId>
       <scope>provided</scope>
</dependency>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

- Crear directorio:
  - /src/main/webapp/WEB-INF/vistas
- Editar src/main/resources/application.properties:

```
spring.mvc.view.prefix=/WEB-INF/vistas/
spring.mvc.view.suffix=.jsp
```


Modelos de datos





https://docs.spring.io/spring-framework/docs/current/spring-framework-reference/core.htm

IOC CON SPRING CORE

Inversión de Control

- Inversión de control (Inversion of Control en inglés, IoC) es un concepto junto a unas técnicas de programación:
  - en las que el flujo de ejecución de un programa se invierte respecto a los métodos de programación tradicionales,
  - en los que la interacción se expresa de forma imperativa haciendo llamadas a procedimientos (procedure calls) o funciones.
- Tradicionalmente el programador especifica la secuencia de decisiones y procedimientos que pueden darse durante el ciclo de vida de un programa mediante llamadas a funciones.
- En su lugar, en la inversión de control se especifican respuestas deseadas a sucesos o solicitudes de datos concretas, dejando que algún tipo de entidad o arquitectura externa lleve a cabo las acciones de control que se requieran en el orden necesario y para el conjunto de sucesos que tengan que ocurrir.

Inyección de Dependencias

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

Introducción

- Spring proporciona un contenedor encargado de la inyección de dependencias (Spring Core Container).
- Este contenedor nos posibilita inyectar unos objetos sobre otros.
- Para ello, los objetos deberán ser simplemente JavaBeans.
- La inyección de dependencias será bien por constructor o bien por métodos setter.
- La configuración podrá realizarse bien por anotaciones Java o mediante un fichero XML (XMLBeanFactory).
- Para la gestión de los objetos tendrá la clase (BeanFactory).
- Todos los objetos serán creados como singletons sino se especifica lo contrario.

Modulo de dependencias

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

```
<beans xmlns:aop="http://www.springframework.org/schema/aop"
       xmlns:context="http://www.springframework.org/schema/context"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xmlns="http://www.springframework.org/schema/beans"
       xsi:schemalocation="http://www.springframework.org/schema/beans
       http://www.springframework.org/schema/beans/spring-beans-2.5.xsd
       http://www.springframework.org/schema/aop
       http://www.springframework.org/schema/aop/spring-aop-2.5.xsd
       http://www.springframework.org/schema/context
       http://www.springframework.org/schema/context/spring-context-2.5.xsd">
      <context:component-scan base-package="es.miEspacio.ioc.services">
      </context:component-scan>
</beans>
```

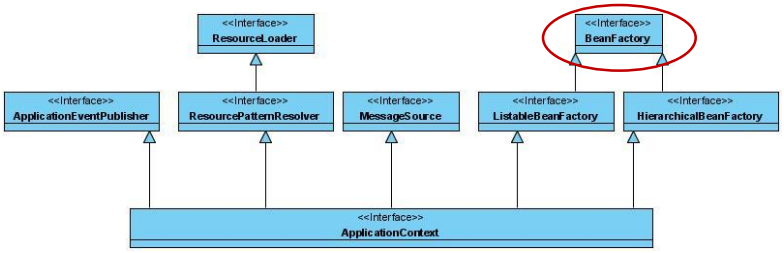
Beans

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

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

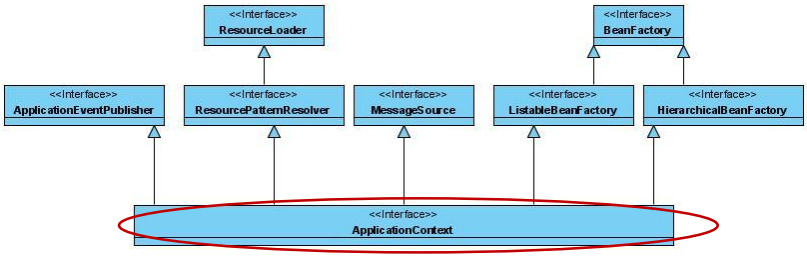

Bean factory

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



Application Context

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



Uso de la inyección de dependencias

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

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

- Muestra:
  - Este es un servicio...

Anotaciones IoC

- Autodescubrimiento
  - @Scope
  - @Component
  - @Repository
  - @Service
  - @Controller
- Personalización
  - @Configuration
  - @Bean
- Inyección
  - @Autowire (@Inject)
  - @Qualifier (@Named)
  - @Value
  - @PropertySource
  - @~~Required~~
  - @Resource
- Otras
  - @PostConstruct
  - @PreDestroy

Esterotipos

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

Alcance

- Un aspecto importante del ciclo de vida de los Beans es si el contenedor creara una única instancia o tantas como ámbitos sean necesarios.
  - prototype: No reutiliza instancias, genera siempre una nueva instancia. @Scope("prototype")
  - singleton: (Predeterminado) Instancia única para todo el contenedor Spring IoC. @Scope("singleton") @Singleton
  - Adicionalmente, en el contexto de un Spring Web ApplicationContext: @RequestScope @SessionScope @ApplicationScope
    - request: Instancia única para el ciclo de vida de una sola solicitud HTTP. Cada solicitud HTTP tiene su propia instancia única.
    - session: Instancia única para el ciclo de vida de cada HTTP Session.
    - application: Instancia única para el ciclo de vida de un ServletContext.
    - websocket: Instancia única para el ciclo de vida de un WebSocket.

Inyección

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

Inyección

- La inyección se realiza con la anotación @Autowire:
  - En atributos:
    ```
    @Autowire
    private MyBeans myBeans;
    ```
  - En propiedades (setter):
    ```
    @Autowire
    public void setMyBeans(MyBeans value) { … }
    ```
  - En constructores
- Por defecto la inyección es obligatoria, se puede marcar como opcional en cuyo caso si no encuentra el Bean inyectará un null.
  ```
  @Autowire(required=false) private MyBeans myBeans;
  ```
- Se puede completar @Autowire con la anotación @Lazy para inyectar un proxy de resolución lenta.

Inyección

- Con @Qualifier (@Named) se pueden agrupar o cualificar los beans asociándoles un nombre:

```
public interface MyInterface { ... }

@Component
@Qualifier("old")
public class MyInterfaceImpl implements MyInterface { ... }

@Qualifier("new")
public class MyNewInterfaceImpl implements MyInterface { ... }

@Autowired(required=false)
@Qualifier("new")
private MyInterface srv;
```

Acceso a ficheros de propiedades

- Localización (fichero .properties, .yml, .xml):
  - Por defecto: src/main/resources/application.properties
  - En la carpeta de recursos src/main/resources:
    @PropertySource("classpath:my.properties")
  - En un fichero local:
    @PropertySource("file://c:/cng/my.properties")
  - En una URL:
    @PropertySource("http://myserver/application.properties")
- Acceso directo:
  @Value("${spring.datasource.username}") private String name;
- Acceso a través del entorno:
  @Autowired private Environment env;
  env.getProperty("spring.datasource.username")

Ciclo de Vida

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

Configuración por código

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

```
public class MyBean { … }

@Configuration
public class MyConfig {
    @Bean
    @Scope("prototype")
    public MyBean myBean() { … ]
```


Doble herencia

- Se crea el interfaz con la funcionalidad deseada:

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

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

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

Cliente

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

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

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

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

@Autowired establece que deben resolverse los parámetros mediante DI.

Anotaciones estándar JSR-330

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

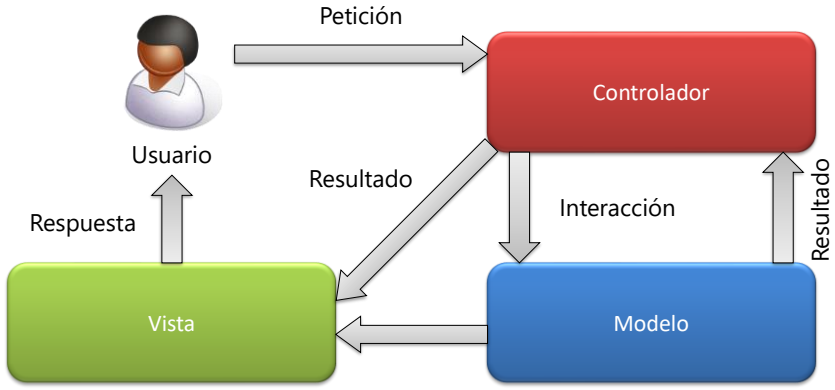
Anotaciones Spring	Anotaciones Estándar (javax.inject.*) JSR-330
@Autowired	@Inject
@Component	@Named / @ManagedBean
@Scope("singleton")	@Singleton
@Qualifier	@Qualifier / @Named
@Value	-
@Required	-
@Lazy	-

INTRODUCCIÓN A SPRING MVC

Spring Web MVC

- Spring Web MVC es el marco web original creado para dar soporte a las aplicaciones web de Servlet-Stack basadas en la API de Servlet y desplegadas en los contenedores de Servlet. Está incluido Spring Framework desde el principio.
- El Modelo Vista Controlador (MVC) es un patrón de arquitectura de software (presentación) que separa los datos y la lógica de negocio de una aplicación del interfaz de usuario y del módulo encargado de gestionar los eventos y las comunicaciones.
- Este patrón de diseño se basa en las ideas de reutilización de código y la separación de conceptos, características que buscan facilitar la tarea de desarrollo de aplicaciones, prueba y su posterior mantenimiento.
- Para todo tipo de sistemas (Escritorio, Web, Movil, …) y de tecnologías (Java, Ruby, Python, Perl, Flex, SmallTalk, .Net …)

El patrón MVC



El patrón MVC

M Modelo
- Representación de los **datos del dominio**
- Lógica de **negocio**
- Mecanismos de **persistencia**

V Vista
- **Interfaz** de usuario
- Incluye elementos de **interacción**

C Controlador
- **Intermediario** entre Modelo y Vista
- **Mapea acciones** de usuario → acciones del Modelo
- **Selecciona** las vistas y les **suministra** información

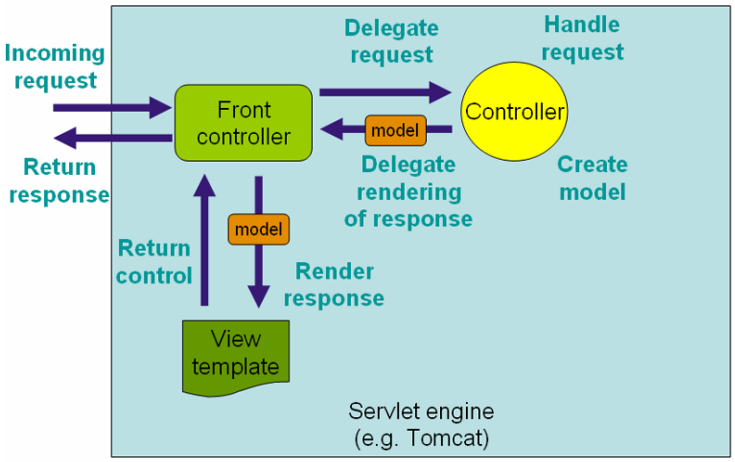
Características de Spring MVC

- Facilidad de mantenimiento.
- Separación limpia entre Controladores, Modelos y Vistas. Lo cual, a su vez, facilita el desarrollo en equipo disciplinado.
- Independencia en la Vista de la aplicación.
- Múltiples tecnologías a la hora de desarrollar las vistas.
- Facilidad a la hora de testear la aplicación.
- Binding y Validaciones.
- Facilidad de trabajar con los componentes gracias a la implementación del patrón de diseño DI "Dependency Injection"

Arquitectura

- El framework Spring MVC está diseñado en base a un DispatcherServlet, que se encarga de distribuir (como su nombre indica) las peticiones a distintos controladores.
- Permite configurar el mapeo de los controladores y resolución de vistas, excepciones, localización, zona horaria y temas.
- Los controladores, anotados con @Controller, son los responsables de preparar un modelo y elegir el nombre de la vista, pero pueden responder directamente y completar la petición si se desea.
- El modelo es una interfaz que permite abstraerse del tipo de tecnología que se está utilizando para la vista, transformándose al formato adecuado cuando se solicita.
- Las vistas pueden generarse utilizando tecnologías como JSP, Velocity, Freemaker o directamente a través de estructuras como XML o JSON.

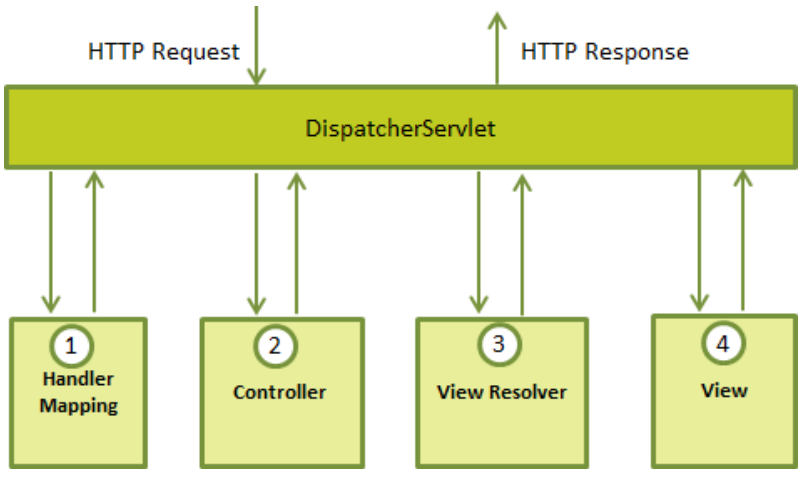
Procesamiento de una solicitud



Procesamiento de una solicitud

- Las peticiones pasan a través de un servlet que actúa de Front Controller (DispatcherServlet).
- El DispatcherServlet consulta a un HandlerMapping para decidir a que controlador le pasa la petición. Usa la URL de la solicitud para decidir.
- El controlador procesa la petición, accede a la lógica de negocio para obtener los resultados y selecciona la vista para presentar el resultado.
  - Para que el controlador no esté acoplado a la vista, se devuelve un identificador lógico (nombre) de vista.
  - Los resultados se encapsulan en un modelo (colección Map)
  - Devuelve al DispatcherServlet un objeto que encapsula ambos: ModelAndView.
- El DispatcherServlet utiliza un ViewResolver para resolver el nombre en una vista concreta
  - Normalmente un JSP, pero se soportan otros Thymeleaf, FreeMarker, XSLT, …
- El DispatcherServlet utiliza la vista para mostrar el modelo al usuario.
  - Puede ser HTML, PDF, XLS, etc.,

Procesamiento de una solicitud



Componentes

- HandlerMapping
  - Asigna una solicitud a un controlador junto con una lista de interceptores para el procesamiento previo y posterior. El mapeo se basa en diferentes criterios, cuyos detalles varían según la implementación HandlerMapping.
  - Las dos implementaciones principales de HandlerMapping son RequestMappingHandlerMapping (basada en las anotaciones @RequestMapping de los métodos) y SimpleUrlHandlerMapping (se define una tabla que asocia URLs a controladores).
- HandlerAdapter
  - Ayuda al DispatcherServlet a invocar al controlador asignado a una solicitud, independientemente de cómo se invoque realmente el controlador. Por ejemplo, invocar un controlador anotado requiere resolver anotaciones.
  - El propósito principal del HandlerAdapter es liberar al DispatcherServlet de tales detalles.

Componentes

- Los Resolver son los encargados de solventar diferentes aspectos a la hora de presentar la respuesta:
  - ViewResolver
    - Resolver la cadena devuelta desde un controlador con el nombre de vista convirtiéndola en una vista real View que procesa la respuesta.
  - HandlerExceptionResolver
    - Estrategia para resolver las excepciones, posiblemente asignándolas a controladores, a vistas de error HTML u otros destinos.
  - LocaleResolver, LocaleContextResolver
    - Resuelva que Locale y posiblemente la zona horaria está utilizando un cliente, para poder ofrecer vistas internacionalizadas.
  - ThemeResolver
    - Resuelve los temas que la aplicación web puede usar, por ejemplo, para ofrecer diseños personalizados.
  - MultipartResolver
    - Abstracción para analizar una solicitud de varias partes (por ejemplo, la carga de un archivo desde formulario) con la ayuda de alguna biblioteca.

Configuración

- Con Spring Boot la configuración se puede realizar en application.properties:

  spring.mvc.view.prefix=/WEB-INF/vistas/

  spring.mvc.view.suffix=.jsp
- En un entorno Servlet 3.0+, existe la opción de configurar el contenedor Servlet mediante programación como alternativa o en combinación con un archivo web.xml.
- La interfaz WebMvcConfigurer define por defecto (Java 8) los métodos que permiten cambiar la configuración MVC de Spring, basta con crear una clase anotada con @Configuration que implemente la interfaz y sobrecargar los métodos que controlen las configuraciones que se desean personalizar.
- El método addViewControllers permite mapear controladores a las URLs dadas para que renderizen la vista dependiendo del nombre dado.

Configuración

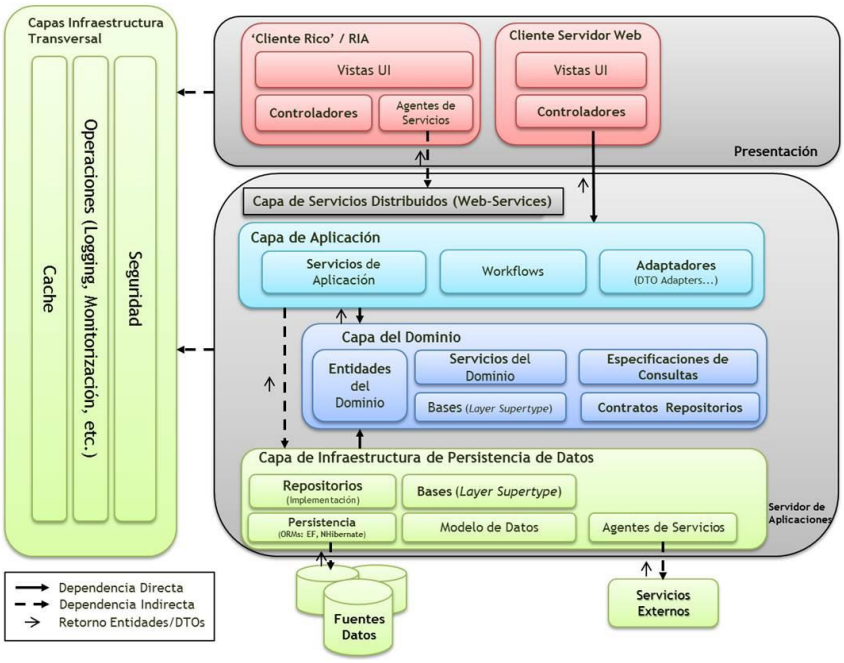
```
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import org.springframework.web.servlet.ViewResolver;
import org.springframework.web.servlet.config.annotation.ViewControllerRegistry;
import org.springframework.web.servlet.config.annotation.WebMvcConfigurer;
import org.springframework.web.servlet.view.InternalResourceViewResolver;

@Configuration
public class AppMvcConfig implements WebMvcConfigurer {
	@Bean
	public ViewResolver viewResolver() {
		InternalResourceViewResolver resolver = new InternalResourceViewResolver();
		resolver.setPrefix("/WEB-INF/vistas/");
		resolver.setSuffix(".jsp");
		return resolver;
	}
	@Override
	public void addViewControllers(ViewControllerRegistry registry) {
		registry.addViewController("/home").setViewName("home");
	}

}
```


ACCESO A DATOS CON SPRING DATA

Spring Data

- Spring Framework ya proporcionaba soporte para JDBC, Hibernate, JPA o JDO, simplificando la implementación de la capa de acceso a datos, unificando la configuración y creando una jerarquía de excepciones común para todas ellas.
- Spring Data es un proyecto (subproyectos) de SpringSource cuyo propósito es unificar y facilitar el acceso a distintos tipos de tecnologías de persistencia, tanto a bases de datos relacionales como a las del tipo NoSQL.
- Spring Data viene a cubrir el soporte necesario para distintas tecnologías de bases de datos NoSQL integrándolas con las tecnologías de acceso a datos tradicionales, simplificando el trabajo a la hora de crear las implementaciones concretas.
- Con cada tipo de tecnología de persistencia, los DAOs (Data Access Objects) ofrecen las funcionalidades típicas de CRUD para objetos de dominio propios, métodos de búsqueda, ordenación y paginación. Spring Data proporciona interfaces genéricas para estos aspectos (CrudRepository, PagingAndSortingRepository) e implementaciones específicas para cada tipo de tecnología de persistencia.

Arquitectura N-Capas con Orientación al Dominio



Modelos: Entidades

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

Anotaciones JPA

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

Asociaciones

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

Asociaciones

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

Asociaciones

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

Cascada

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

Mapeo de Herencia

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

DTO

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

Lombok

https://projectlombok.org/

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

```
public @Data @AllArgsConstructor class MyDTO {
        private long id;
        private String name;
}
```

- Es necesario agregar las bibliotecas al proyecto.

ModelMapper

http://modelmapper.org/

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

```
ModelMapper modelMapper = new ModelMapper();
OrderDTO orderDTO = modelMapper.map(order, OrderDTO.class);
```

Serialización Jackson

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

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

- El proceso de serialización y des serialización se puede controlar declarativamente mediante anotaciones:
  https://github.com/FasterXML/jackson-annotations

Serialización Jackson

- **@JsonProperty**: indica el nombre alternativo de la propiedad en JSON.

```
@JsonProperty("name") public String getTheName() { ... }
@JsonProperty("name") public void setTheName(String name) { ... }
```

- **@JsonFormat**: especifica un formato para serializar los valores de fecha/hora.

```
@JsonFormat(shape = JsonFormat.Shape.STRING, pattern = "dd-MM-yyyy hh:mm:ss")
public Date eventDate;
```

- **@JsonIgnore**: marca que se ignore una propiedad (nivel miembro).

```
@JsonIgnore public int id;
```

Serialización Jackson

- **@JsonIgnoreProperties**: marca que se ignore una o varias propiedades (nivel clase).

```
@JsonIgnoreProperties({ "id", "ownerName" })
@JsonIgnoreProperties(ignoreUnknown=true)
public class Item {
```

- **@JsonInclude**: se usa para incluir propiedades con valores vacíos/nulos/ predeterminados.

```
@JsonInclude(Include.NON_NULL)
public class Item {
```

- **@JsonAutoDetect**: se usa para anular la semántica predeterminada qué propiedades son visibles y cuáles no.

```
@JsonAutoDetect(fieldVisibility = Visibility.ANY)
public class Item {
```


Serialización Jackson

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

```
public class Views {
   public static class Partial {}
   public static class Complete extends Partial {}
}
public class Item {
   @JsonView(Views.Partial.class)
   public int id;
   @JsonView(Views.Partial.class)
   public String itemName;
   @JsonView(Views.Complete.class)
   public String ownerName;
}

String result = new ObjectMapper().writerWithView(Views.Partial.class)
      .writeValueAsString(item);
```

Serialización Jackson

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

```
@JsonFilter("ItemFilter")
public class Item {
   public int id;
   public String itemName;
   public String ownerName;
}

FilterProvider filters = new SimpleFilterProvider().addFilter("ItemFilter",
   SimpleBeanPropertyFilter.filterOutAllExcept("id", "itemName"));
MappingJacksonValue mapping = new
   MappingJacksonValue(dao.findAll());
mapping.setFilters(filters);
return mapping;
```


Serialización Jackson

- **@JsonManagedReference** y **@JsonBackReference**: se utilizan para manejar las relaciones maestro/detalle marcando la colección en el maestro y la propiedad inversa en el detalle (múltiples relaciones requieren asignar nombres únicos).

```
@JsonManagedReference
public User owner;
@JsonBackReference
public List<Item> userItems;
```

- **@JsonIdentityInfo**: indica la identidad del objeto para evitar problemas de recursión infinita.

```
@JsonIdentityInfo(
  generator = ObjectIdGenerators.PropertyGenerator.class,
  property = "id")
public class Item {
   public int id;
```

Serialización XML (JAXB)

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

Anotaciones principales (JAXB)

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

Validaciones

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

```
public ResponseEntity<Object> create(@Valid @RequestBody Persona item)
```

- Para realizar la validación manualmente:

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


Validaciones

@Null : Comprueba que el valor anotado es null

@NotNull : Comprueba que el valor anotado no sea null

@NotEmpty : Comprueba si el elemento anotado no es nulo ni está vacío

@NotBlank : Comprueba que la secuencia de caracteres anotados no sea nula y que la longitud recortada sea mayor que 0. La diferencia @NotEmptyes que esta restricción solo se puede aplicar en secuencias de caracteres y que los espacios en blanco finales se ignoran.

@AssertFalse : Comprueba que el elemento anotado es falso.

@AssertTrue : Comprueba que el elemento anotado es verdadero

Validaciones

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

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

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

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

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

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

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

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

Validaciones

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

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

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

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

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

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

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

Repositorio

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

Repositorio

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

```
@Repository
public interface ProfesorRepository extends JpaRepository<Profesor, Long> {}
@Autowired
private ProfesorRepository repository;
```

Repositorio

- El interfaz puede ser ampliado con nuevos métodos que serán implementados por Spring:
  - Derivando la consulta del nombre del método directamente.
  - Mediante el uso de una consulta definida manualmente.
- La implementación se realizará mediante la decodificación del nombre del método, dispone de una sintaxis especifica para crear dichos nombre:

```
List<Profesor> findByNombreStartingWiths(String nombre);
List<Profesor> findByApellido1AndApellido2OrderByEdadDesc( String
        apellido1, String apellido2);
List<Profesor> findByTipoIn(Collection<Integer> tipos);

int deleteByEdadGreaterThan(int valor);
```


Repositorio

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

Repositorio

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

Repositorio

Palabra clave	Muestra	Fragmento de JPQL
After	findByStartDateAfter	… where x.startDate > ?1
Before	findByStartDateBefore	… where x.startDate < ?1
IsNull	findByAgeIsNull	… where x.age is null
IsNotNull, NotNull	findByAge(Is)NotNull	… where x.age not null
Like	findByFirstnameLike	… where x.firstname like ?1
NotLike	findByFirstnameNotLike	… where x.firstname not like ?1
StartingWith	findByFirstnameStartingWith	… where x.firstname like ?1 (parámetro enlazado con % anexado)
EndingWith	findByFirstnameEndingWith	… where x.firstname like ?1(parámetro enlazado con % antepuesto)

Repositorio

P. Clave	Muestra	Fragmento de JPQL
Containing	findByFirstnameContaining	… where x.firstname like ?1(parámetro enlazado entre %)
OrderBy	findByAgeOrderByLastnameDesc	… where x.age = ?1 order by x.lastname desc
Not	findByLastnameNot	… where x.lastname <> ?1
In	findByAgeIn(Collection<Age> ages)	… where x.age in ?1
NotIn	findByAgeNotIn(Collection<Age> ages)	… where x.age not in ?1
True	findByActiveTrue()	… where x.active = true
False	findByActiveFalse()	… where x.active = false
IgnoreCase	findByFirstnameIgnoreCase	… where UPPER(x.firstame) = UPPER(?1)

Repositorio

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

Repositorio

- Mediante consultas JPQL:

```
@Query("from Profesor p where p.edad > 67")
List<Profesor> findJubilados();

@Modifying
@Query("delete from Profesor p where p.edad > 67")
List<Profesor> deleteJubilados();
```

- Mediante consultas SQL nativas:

```
@Query("select * from Profesor p where p.edad between ?1 and ?2", nativeQuery=true)
List<Profesor> findActivos(int inicial, int final);
```


Transacciones

- Por defecto, los métodos CRUD en las instancias del repositorio son transaccionales. Para las operaciones de lectura, el indicador readOnly de configuración de transacción se establece en true para optimizar el proceso. Todos los demás se configuran con un plano @Transactional para que se aplique la configuración de transacción predeterminada.
- Cuando se van a realizar varias llamadas al repositorio o a varios repositorios se puede anotar con @Transactional el método para que todas las operaciones se encuentren dentro de la misma transacción.

```
@Transactional
  public void create(Pago pago) { … }
```

- Para que los métodos de consulta sean transaccionales:

```
@Modifying
 @Transactional
 @Query("delete from User u where u.active = false")
 void deleteInactiveUsers();
```

Servicio

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

Patrón Agregado (Aggregate)

- Una Agregación es un grupo de objetos asociados que deben tratarse como una unidad a la hora de manipular sus datos.
- El patrón Agregado es ampliamente utilizado en los modelos de datos basados en Diseños Orientados al Dominio (DDD).
- Proporciona un forma de encapsular nuestras entidades y los accesos y relaciones que se establecen entre las mismas de manera que se simplifique la complejidad del sistema en la medida de lo posible.
- Cada Agregación cuenta con una Entidad Raíz (root) y una Frontera (boundary):
  - La Entidad Raíz es una Entidad contenida en la Agregación de la que colgarán el resto de entidades del agregado y será el único punto de entrada a la Agregación.
  - La Frontera define qué está dentro de la Agregación y qué no.
- La Agregación es la unidad de persistencia, se recupera toda y se almacena toda.

CONTROLADORES

Controladores

- Son clases Java anotadas con @Controller, que no requieren una herencia o interface especifico, con métodos que reciben y responden a las peticiones de los clientes.

```
@Controller()
public class PaisController {
        @Autowired
        private PaisRepository paisRepository;
```

- Los métodos de la clase que interactúan con el cliente pueden llevar la anotación @RequestMapping, con la subruta y el RequestMethod.

```
@RequestMapping(value = "/paises", method = RequestMethod.GET)
public String getAll(Model model) {
    model.addAttribute("listado", paisRepository.findAll());
    return "pais/list"; // view name
}
```

Métodos de mapeo

- La anotación @RequestMapping permite asignar solicitudes a los métodos de los controladores.
- Tiene varios atributos para definir: URL, método HTTP, parámetros de solicitud, encabezados y tipos de medios.
- Se puede usar al nivel de clase para expresar asignaciones compartidas o al nivel de método para limitar a una asignación de endpoint específica.
- También hay atajos con el método HTTP predefinido:
  - @GetMapping
  - @PostMapping
  - @PutMapping
  - @DeleteMapping
  - @PatchMapping

Patrones de URI

- Establece que URLs serán derivadas al controlador.
- Puede asignar solicitudes utilizando los siguientes patrones globales y comodines:
  - ? cualquier carácter
  - * cero o más caracteres dentro de un segmento de ruta
  - ** cero o más segmentos de ruta
- También puede declarar variables en la URI y acceder a sus valores con anotando con @PathVariable los parámetros, debe respetarse la correspondencia de nombres:

```
@GetMapping("/owners/{ownerId}/pets/{petId}")
public String findPet(@PathVariable Long ownerId, @PathVariable Long
petId) {
   // ...
}
```

Restricciones

- cosumes: formatos MIME permitidos del encabezado Content-type

```
@PostMapping(path = "/pets", consumes = "application/json")
public String addPet(@RequestBody Pet pet) {
```

- produces : formatos MIME permitidos del encabezado Accept

```
@GetMapping(path = "/pets/{petId}", produces =
"application/json;charset=UTF-8")
@ResponseBody
public String getPet(@PathVariable String petId) {
```

- params: valores permitidos de los QueryParams
- headers: valores permitidos de los encabezados

```
@GetMapping(path = "/pets/{petId}", params = "myParam=myValue",
      headers = "myHeader=myValue")
public String findPet(@PathVariable String petId) {
```


HanderMapping

- La interface HandlerMapping es utilizada por el DispatcherServlet para determinar cual es el controlador que debe manejar una petición HTTP, este componente analiza la URL de la petición y determina a que controlador se debe llamar para responder a la misma, el HandlerMapping utilizado por defecto es DefaultAnnotationHandlerMapping.
- El BeanNameUrlHandlerMapping mapea una URL usando el nombre del bean controlador, es decir la URL a la que responder un controlador estará determinada por su nombre, indicando el nombre de un bean en la anotación, si este no aparece se utilizara el nombre del método con la letra inicial en minúscula.

```
@Bean
public HandlerMapping handlerMapping() { return new BeanNameUrlHandlerMapping(); }
@Bean("/inicio.html")
public HomeController homeController() { return new HomeController(); }
```

- El SimpleUrlHandlerMapping utiliza una colección para almacenar las URL y sus correspondientes controladores, mediante un objeto Properties:

```
@Bean
public HandlerMapping handlerMapping() {
   Properties urlMaps = new Properties();
   urlMaps.put("/index.html", "homeController");
   // … Resto de mapeos
   SimpleUrlHandlerMapping handler = new SimpleUrlHandlerMapping();
   handler.setMappings(urlMaps);
   return handler;
}
```

Inyección de Parámetros

- El API decodifica la petición e inyecta los datos como parámetros en el método.
- Es necesario anotar los parámetros para indicar la fuente del dato a inyectar.
- En las anotaciones será necesario indicar el nombre del origen en caso de no existir correspondencia de nombres con el de los parámetros.
- El tipo de origen, en la mayoría de los casos, es String que puede discrepar con los tipos de los parámetros, en tales casos, la conversión de tipo se aplica automáticamente en función de los convertidores configurados.
- Por defecto los parámetros son obligatorios, se puede indicar que sean opcionales, se inicializaran a null si no reciben en la petición salvo que se indique el valor por defecto:
  - @RequestParam(required=false, defaultValue="1")

Inyección de Parámetros

Anotación	Descripcción
@PathVariable	Para acceder a las variables de la plantilla URI.
@MatrixVariable	Para acceder a pares nombre-valor en segmentos de ruta URI.
@RequestParam	Para acceder a los parámetros de solicitud del Servlet (QueryString o Form), incluidos los archivos de varias partes. Los valores de los parámetros se convierten al tipo de argumento del método declarado.
@RequestHeader	Para acceder a las cabeceras de solicitud. Los valores de encabezado se convierten al tipo de argumento del método declarado.
@CookieValue	Para el acceso a las cookies. Los valores de las cookies se convierten al tipo de argumento del método declarado.

Inyección de Parámetros

Anotación	Descripcción
@RequestBody	Para acceder al cuerpo de la solicitud HTTP. El contenido del cuerpo se convierte al tipo de argumento del método declarado utilizando implementaciones HttpMessageConverter.
@RequestPart	Para acceder a una parte en una solicitud multipart/form-data, convertir el cuerpo de la parte con un HttpMessageConverter.
@ModelAttribute	Para acceder a un atributo existente como modelo (instanciado si no está presente) con enlace de datos y validación aplicada.
@SessionAttribute	Para acceder a cualquier atributo de sesión, a diferencia de los atributos de modelo almacenados en la sesión como resultado de una declaración @SessionAttributes de nivel de clase .
@RequestAttribute	Para acceder a los atributos de solicitud.

Modelo

- El interfaz Model actúa como intermediario en el traspaso de datos entre el controlador y la vista.
- El objeto model se inyecta a través del método:

  public String getAll(Model model) {
- Implementa una colección Map<String, Object> que permite añadir cualquier tipo de datos asociado al literal identificador.
- Para añadir contenidos al model:

  model.addAttribute("name", "mundo");
- En la vista se invoca directamente a través del identificador:

  <h1>Hola ${name}</h1>

ModelAndView

- La clase ModelAndView encapsula en un objeto la vista junto con el modelo con los datos para la vista.
- El método debe crear, rellenar y devolver un objeto de tipo ModelAndView:
- El objeto model se inyecta a través del método:

```
public ModelAndView getAll() {
      ModelAndView mv = new ModelAndView();
      // …
      return mv;
}
```

- Para añadir contenidos al model:

```
mv.addObject("name", "mundo");
```

- Seleccionar la vista:

```
mv.setViewName("list");
```


Inyección de Parámetros

```
// http://localhost:8080/params/1?nom=kk
@GetMapping("/params/{id}")
public String cotilla(
      @PathVariable String id,
      @RequestParam String nom,
      @RequestHeader("Accept-Language") String language,
      @CookieValue("JSESSIONID") String cookie,
      Model model) {
   StringBuilder sb = new StringBuilder();
   sb.append("id: " + id + "<br>");
   sb.append("nom: " + nom + "<br>");
   sb.append("language: " + language + "<br>");
   sb.append("cookie: " + cookie + "<br>");
   model.addAttribute("message", sb.toString());
   return "message";
}
```

Inyecciones adicionales

- Se definen los parámetros de los tipos adecuados Spring inyectara los objetos indicados.
- Para acceder a los valores originales de los parámetros anotados:
  - WebRequest, NativeWebRequest, MultipartRequest, ServletRequest, ServletResponse, MultipartHttpServletRequest, HttpSession, SessionStatus, HttpMethod,
- Para acceder a la configuración regional de la solicitud actual y la zona horaria asociada
  - java.util.Locale, java.util.TimeZone
- Para acceder al usuario autenticado actual
  - java.security.Principal
- Para acceder al cuerpo de la solicitud o de la respuesta sin procesar
  - java.io.InputStream, java.io.Reader, java.io.OutputStream, java.io.Writer
- Para acceder al modelo
  - java.util.Map, org.springframework.ui.Model, org.springframework.ui.ModelMap
- Para acceder a los errores de validación y enlace de datos
  - Errors, BindingResult
- Para preparar una URL relacionada con la solicitud actual
  - UriComponentsBuilder

Enlace de datos

- Puede usar la anotación @ModelAttribute (opcional) en un argumento del método para acceder a un atributo del modelo o hacer que se cree una instancia si no está presente.
- El atributo del modelo también está superpuesto con valores de los parámetros de solicitud del Servlet HTTP cuyos nombres coinciden con los nombres de los campos. Esto se conoce como enlace de datos y nos evita tener que lidiar con el análisis y la conversión de parámetros de consulta individuales y campos de formulario.
- La clase WebDataBinder hace coincidir los nombres de parámetros de solicitud de Servlet (parámetros de consulta y campos de formulario) con los nombres de campo del objeto destino, después de que se aplique la conversión de tipo en caso de ser necesario.
- El enlace de datos puede dar lugar a errores, lanza una BindException antes de ejecutar el método. Para evitar la excepción y recoger los errores hay que agregar un argumento BindingResult inmediatamente al lado del @ModelAttribute:

```
addPost(@ModelAttribute("elemento") Region item, BindingResult result)
```

@RequestBody

- Puede utilizar la anotación @RequestBody para que el cuerpo de la solicitud se lea y se deserialice a parámetro a través de HttpMessageConverter.

```
@PostMapping("/accounts")
public void handle(@RequestBody Account account) {
```

- Se puede usar convertidores de mensajes para configurar o personalizar la conversión de mensajes.
- Al igual que con @ModelAttribute, se puede usar @RequestBody en combinación con la anotación javax.validation.Valid o la de Spring @Validated, ambas causan que se aplique la validación estándar de Bean. De forma predeterminada, los errores de validación causan una MethodArgumentNotValidException, que se convierte en una respuesta 400 (BAD_REQUEST). Alternativamente, puede manejar los errores de validación localmente dentro del controlador a través de un argumento Errors o BindingResult:

```
@PostMapping("/region")
public void handle(@Valid @RequestBody Region item, BindingResult result) {
```


Respuesta

- La respuesta por defecto es una cadena con el nombre de la vista.
- Se puede devolver la encapsulación de la vista y el modelo en un objeto ModelAndView, o directamente un View con la instancia de la vista.
- Con la anotación @ModelAttribute, o los tipos devueltos java.util.Map o Model, la respuesta es el modelo.
- Si es void se delega en la anotación @ResponseStatus establecer el HTTP Status Code resultante.
- Con la anotación @ResponseBody y los tipos HttpEntity y ResponseEntity el valor de retorno especifica el cuerpo o la respuesta completa (incluidos los encabezados y el cuerpo de HTTP).

RequestToViewNameTranslator

- Normalmente, se debe devolver explícitamente el nombre de la vista desde el método del controlador, pero en su lugar podemos aprovechar la resolución automática de nombres proporcionada por una implementación de RequestToViewNameTranslator, que es capaz de traducir un HttpServletRequest entrante en un nombre de vista lógica cuando no se devuelve explícitamente un nombre de vista desde el controlador.
- DefaultRequestToViewNameTranslator es la única implementación de esta interfaz que tiene Spring, que transforma la URI de solicitud en un nombre de vista. Tiene las siguientes propiedades configurables:
  - prefix: una cadena que antepone al nombre de vista generado. El valor predeterminado es "".
  - suffix: una cadena que se añade al nombre de vista generado. El valor predeterminado es "".
  - separator: una cadena que separa las partes de URI. El valor predeterminado es "/".
  - stripLeadingSlash: un valor booleano que especifica si se debe eliminar una barra inicial, el valor predeterminado es verdadero
  - stripTrailingSlash: un valor booleano que especifica si se debe eliminar la barra diagonal final, el valor predeterminado es verdadero.
  - stripExtension: un valor booleano que especifica si se debe eliminar la extensión de archivo; el valor predeterminado es verdadero.

  @GetMapping("page1") → src/main/webapp/WEB-INF/views/page1.jsp
  @GetMapping("data/page2") → src/main/webapp/WEB-INF/views/data/page2.jsp

Respuestas sin vistas

- La anotación @ResponseBody en un método indica que el retorno será serializado en el cuerpo de la respuesta a través de un HttpMessageConverter.

```
@PostMapping("/invierte")
@ResponseBody
public Punto body(@RequestBody Punto p) {
      int x = p.getX();
      p.setX(p.getY());
      p.setY(x);
      return p;
}
```

- El código de estado de la respuesta se puede establecer con la anotación @ResponseStatus:

```
@PostMapping
@ResponseStatus(HttpStatus.CREATED)
public void add(@RequestBody Punto p) { … }
```

Respuesta personalizada

- La clase ResponseEntity permite agregar estado y encabezados a la respuesta (no requiere la anotación @ResponseBody).

```
@GetMapping(value="/pais")
public ResponseEntity<List<Pais>> getAll(){
    return new ResponseEntity<List<Pais>>(
            paisRepository.findAll(),
            HttpStatus.OK);
}
```

- La clase ResponseEntity dispone de builder para generar la respuesta:

```
return ResponseEntity.ok().eTag(etag).build(body);
```


Redirecciones

- El prefijo especial “redirect:” en el nombre de una vista permite realizar una redirección.
  - return "redirect:/list";
- Los UrlBasedViewResolver (y sus subclases) reconocen esto como una instrucción de que se necesita una redirección.
- El resto de la cadena ya no es el nombre de la vista, es la URL relativa o absoluta de redireccionamiento.
- La redirección genera una respuesta de tipo 302 (Found), el contenido original ha cambiado de sitio de forma temporal.
- Se devuelve una URL que el navegador debe solicitar automáticamente, pero el navegador debe seguir utilizando la URL original en las próximas peticiones.
- El redireccionamiento es fundamental para que el navegador mantenga sincronizada la página que está visualizando con la dirección URL que realizo la petición para poder refrescar la página. Por ejemplo en los POST de los formularios o el borrado de elementos.
- En caso contrario, si no se usa la redirección, cuando se solicite el refresco de la página, el servidor debe se seguir los mismos pasos para generar la misma vista que actualmente esta visualizando el usuario.

Paginación y Ordenación

QueryString	Descripción
page	Número de página en base 0. Por defecto: página 0.
size	Tamaño de página. Por defecto: 20.
sort	Propiedades de ordenación en el formato property,property(,ASC|DESC). Por defecto: ascendente. Hay que utilizar varios sort para diferente direcciones (?sort=firstname&sort=lastname,asc)

```
@GetMapping
public List<Employee> getAll(Pageable pageable) {
        if(pageable.isPaged()) {
                return dao.findAll(pageable).getContent();
        } else
                return dao.findAll();
}
```

- @PageableDefault: Anota a Pageable para establecer los valores por defecto de paginación y ordenación

```
@PageableDefault(size=10, sort = {"firstName", "lastName"}) Pageable pageable.
```

- PageRequest: Permite generar un objeto Pageable:

```
PageRequest.of(0, 10, Sort.by("firstName"));
```


Tratamiento de excepciones.

- Siempre que se produce una excepción no recuperable hay que notificar al usuario y al navegador de la situación anómala.
- El protocolo HTTP cuenta con un conjunto de códigos para notificar el estado de la petición.
- Es necesario traducir la excepción a uno de dichos códigos y, en caso de ser necesario, acompañarlo con un mensaje. En caso contrario todas las excepciones se mostraran como “500 Error interno del servidor”.
- El tratamiento de la excepción se puede realizar en:
  - El método del controlador donde se ha producido.
  - En el controlado, a través de métodos anotados con @ExceptionHandler que intercepten determinadas excepciones producidas en el resto de los métodos de la clase.
  - Globalmente, para todas las excepciones de los diferentes controladores, en clases anotadas con @ControllerAdvice que solo tienen los correspondientes métodos @ExceptionHandler.
  - Globalmente, para todas las excepciones de los diferentes controladores, mediante un controlador de error mapeado a “/error”

Lanzar excepciones.

- El sistema cuenta con una serie de excepciones predefinidas que tienen ya asociados sus correspondientes HTTP Status Code.
- Se puede lanzar cualquier tipo de excepción que estará asociada el HTTP Status Code 500.
- Está disponible la especialización ResponseStatusException que permite indicar el HTTP Status Code y el mensaje asociado.

  throw new ResponseStatusException(HttpStatus.NOT_FOUND);
- Podemos crear nuestras propios herederos de ResponseStatusException que establezcan de forma predeterminada un HTTP Status Code concreto.
  - NotFoundException, BadRequestException, …

Excepciones predefinidas

Exception	HTTP Status Code
HttpRequestMethodNotSupportedException	405 (SC_METHOD_NOT_ALLOWED)
HttpMediaTypeNotSupportedException	415 (SC_UNSUPPORTED_MEDIA_TYPE)
HttpMediaTypeNotAcceptableException	406 (SC_NOT_ACCEPTABLE)
MissingPathVariableException	500 (SC_INTERNAL_SERVER_ERROR)
MissingServletRequestParameterException	400 (SC_BAD_REQUEST)
ServletRequestBindingException	400 (SC_BAD_REQUEST)
ConversionNotSupportedException	500 (SC_INTERNAL_SERVER_ERROR)
TypeMismatchException	400 (SC_BAD_REQUEST)
HttpMessageNotReadableException	400 (SC_BAD_REQUEST)
HttpMessageNotWritableException	500 (SC_INTERNAL_SERVER_ERROR)
MethodArgumentNotValidException	400 (SC_BAD_REQUEST)
MissingServletRequestPartException	400 (SC_BAD_REQUEST)
BindException	400 (SC_BAD_REQUEST)
NoHandlerFoundException	404 (SC_NOT_FOUND)
AsyncRequestTimeoutException	503 (SC_SERVICE_UNAVAILABLE)

Excepciones personalizadas

```
import org.springframework.http.HttpStatus;
import org.springframework.web.server.ResponseStatusException;

public class NotFoundException extends ResponseStatusException {
    public NotFoundException() {
        super(HttpStatus.NOT_FOUND, " ");
    }
    public NotFoundException(String reason) {
        super(HttpStatus.NOT_FOUND, reason);
    }
    public NotFoundException(String reason, Throwable cause) {
        super(HttpStatus.NOT_FOUND, reason, cause);
    }
    public NotFoundException(Throwable cause) {
        super(HttpStatus.NOT_FOUND, " ", cause);
    }
}
```


@ControllerAdvice

```
@ControllerAdvice
public class ApiExceptionHandler {
   @ResponseStatus(HttpStatus.NOT_FOUND)
   @ExceptionHandler({NotFoundException.class})
    public ModelAndView notFoundRequest(HttpServletRequest request, Exception exception) {
      ModelAndView mv = new ModelAndView("errorPage");
      mv.addObject("error", "404 NOT FOUND");
      mv.addObject("message", request.getRequestURI());
      return mv;
   }
   @ResponseStatus(HttpStatus.BAD_REQUEST)
   @ExceptionHandler({ BadRequestException.class, MalformedHeaderException.class,
         FieldInvalidException.class
   })
   public ModelAndView badRequest(Exception exception) {
      ModelAndView mv = new ModelAndView("errorPage");
      mv.addObject("error", "400 BAD_REQUEST");
      mv.addObject("message", exception.getMessage());
       return mv;
   }}
```

Controlador de errores

- Por defecto, Spring Boot asigna “/error” al BasicErrorController que llena el modelo con los atributos de error y luego devuelve 'error' como el nombre de la vista para asignar las páginas de error definidas por la aplicación.
- Para reemplazar BasicErrorController con nuestro propio controlador personalizado que puede asignarse a “/error”, necesitamos implementar la interfaz ErrorController.
- Si inyectamos un WebRequest al método obtendremos un DefaultErrorAttributes con el contexto de error que se podrá mapear a la vista.
- Para acceder a través de HttpServletRequest:

```
code = (int)request.getAttribute("javax.servlet.error.status_code");
msg = (String) request.getAttribute("javax.servlet.error.message");
```

- Adicionalmente se pueden cambiar las opciones por defecto:

```
server.error.whitelabel.enabled=false
server.error.include-stacktrace=always
```


DefaultErrorAttributes

Atributo	Descripción
timestamp	La hora en que se extrajeron los errores
status	El HTTP Status Code
error	La razón del error
exception	El nombre de clase de la excepción raíz
message	El mensaje de excepción
errors	Cualquier ObjectErrors de una BindingResultExcepción
trace	El rastreo de la pila de excepciones
path	La ruta URL donde se generó la excepción

Controlador personalizado

```
@Controller
public class AppErrorController implements ErrorController {
   private final static String ERROR_PATH = "/error";
   @Autowired
   private ErrorAttributes errorAttributes; // Error Attributes in Application

   @RequestMapping(path = ERROR_PATH, produces = "text/html")
   public ModelAndView errorHtml(WebRequest request) {
      return new ModelAndView("/errorPage",
                errorAttributes.getErrorAttributes(request, false));
   }
   @Override
   public String getErrorPath() {
      return ERROR_PATH;
   }
}
```


VISTAS

ViewResolver

- Spring MVC puede generar una gran variedad de vistas, entre ellas: HTML, PDF, XLS, RSS, JSP, JSON, XML, etc., estas son creadas usando diversas tecnologías, como: JSP, Thymeleaf, Velocity, FreeMarker, JasperReport, etc.,
- Spring MVC define las interfaces ViewResolver y View que le permiten representar modelos en un navegador sin tener que vincularlo a una tecnología de vista específica.
- ViewResolver proporciona una asignación entre los nombres de vista y las vistas reales, es decir, obtener el archivo físico que se usará para generar la vistas a partir del nombre lógico devuelto por el controlador
- View aborda la preparación de los datos antes de transferirlos a una tecnología de visualización específica.
- En caso de utilizar Spring Boot, la anotación @EnableAutoConfiguration añade la clase de auto-configuración WebMvcAutoConfiguration que a su vez añade al contexto varios tipos de ViewResolver, dependiendo del resto de módulos integrados en el proyecto.

ViewResolver

- UrlBasedViewResolver
  - Resolución directa de los nombres cuando los nombres lógicos coinciden con los nombres de los recursos de vista de una manera directa, sin la necesidad de asignaciones arbitrarias.
  - InternalResourceViewResolver
    - Resuelve las InternalResourceView (Servlets y JSP) y su subclases como JstlView y TilesView. Puede especificar la clase de vista para todas las vistas generadas por esta resolución utilizando setViewClass(..).
  - ResourceBundleViewResolver
    - Resuelve usando las definiciones de bean especificadas por el nombre base del paquete, utilizando el valor de la propiedad [viewname].(class) como la clase de vista y el valor de la propiedad [viewname].url como la URL de vista.
- FreeMarkerViewResolver
  - Resuelve las FreeMarkerView y las subclases personalizadas de ellos.
- ContentNegotiatingViewResolver
  - Resuelve las vista basada en el nombre del archivo de solicitud o el encabezado Accept.
- AbstractCachingViewResolver
  - El almacenamiento en caché mejora el rendimiento de ciertas tecnologías de visualización.
- XmlViewResolver
  - Acepta un archivo de configuración escrito en XML con la misma DTD que las factorias de beans XML de Spring. El archivo de configuración por defecto es /WEB-INF/views.xml.

BeanNameViewResolver

- Esta clase resuelve la vista usando el nombre de los bean, esto quiere decir que necesitamos tener un bean con el nombre de la vista que indique la ubicación del archivo JSP que será usado para generar dicha vista.
- El nombre del bean se puede establecer en la anotación, si no, se utilizará el nombre del método con la primera letra en minúscula como nombre del bean.

```
@Bean
public ViewResolver viewResolver() {
    return new BeanNameViewResolver();
}
@Bean(name = "home")
public View home() {
   JstlView view = new JstlView();
   view.setUrl("/WEB-INF/views/home.jsp");
   return view;
}
```


XmlViewResolver

- Similar a BeanNameViewResolver, salvo que los beans que se definen en el archivo /WEB-INF/views.xml, el nombre lógico de la vista debe coincidir con el nombre del bean.

```
@Bean
public ViewResolver xmlViewResolver() { return new XmlViewResolver(); }
```

- El archivo /WEB-INF/views.xml se define de la siguiente manera:

```
<?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-4.3.xsd">
   <bean id="home" class="org.springframework.web.servlet.view.JstlView">
      <property name="url" value="/WEB-INF/views/home.jsp"/>
   </bean>
</beans>
```

ResourcBundleViewResolver

- Alternativa a XmlViewResolver que utilizara un archivo .properties que indica como resolver las vistas, el nombre del archivo se establece usando el método setBasename("..."), que debe estar ubicado en la carpeta: src/main/resources.

```
@Bean
public ViewResolver resourceBundleViewResolver() {
   ResourceBundleViewResolver resolver = new ResourceBundleViewResolver();
   resolver.setBasename("views");
   return resolver;
}
```

- Requiere varias entradas por vista:
  - vista.(class) : define la clase usada para generar la vista,
  - vista.url : indica el archivo con la plantilla usado para generar dicha vista.
- El archivo src/main/resources/views.properties tendrá el siguiente contenido:

```
home.(class)=org.springframework.web.servlet.view.JstlView
home.url=/WEB-INF/views/home.jsp
```


InternalResourceViewResolver

- Es el ViewResolver mas utilizado dado que no requiere registro de las vistas, compone el nombre físico de la vista usando un prefijo, el nombre lógico un sufijo. Usando los métodos setPrefix("...") se indica la carpeta donde se almacenan los archivos y con setSuffix("...") se indica la extensión de los archivos.

```
@Bean
public ViewResolver viewResolver() {
   InternalResourceViewResolver resolver = new InternalResourceViewResolver();
   resolver.setPrefix("/WEB-INF/views/");
   resolver.setSuffix(".jsp");
   return resolver;
}
```

- El método setViewClass(...) permite establecer la clase que se utilizara para generar la vista, por defecto utiliza la clase JstlView.
- También se puede configurar a través sobreescritura del método configureViewResolvers del interface WebMvcConfigurer:

```
@Configuration
public class AppMvcConfig implements WebMvcConfigurer {
   @Override
   public void configureViewResolvers(ViewResolverRegistry registry) {
        registry.jsp("/WEB-INF/vistas/", ".jsp");
   }
}
```

Usar varios ViewResolver

- Se puede utilizar mas de un tipo de ViewResolver en la misma aplicación web, pero es necesario establecer una prioridad para el caso en que mas de uno pueda resolver la misma vista lógica.
- A través del método setOrder(int) se establece la prioridad, siendo los valores bajos los más prioritarios y los altos los menos.

```
@Bean
public ViewResolver resourceBundleViewResolver() {
   ResourceBundleViewResolver resolver = new ResourceBundleViewResolver();
   resolver.setBasename("views");
   resolver.setOrder(1);
   return resolver;
}
@Bean
public ViewResolver viewResolver() {
   InternalResourceViewResolver resolver = new InternalResourceViewResolver();
   resolver.setPrefix("/WEB-INF/views/");
   resolver.setSuffix(".jsp");
   resolver.setOrder(2);
   return resolver;
}
```


Configuración

- Se considera buena práctica que todas las plantillas se almacenen en el mismo directorio y sean del mismo tipo, para simplificar los controladores se puede especificar al ViewResolver el prefijo y el sufijo del nombre del archivo ofrecido.
- El directorio src/main/webapp/WEB-INF se encuentran archivos que no forman parte del directorio público de la aplicación, por lo tanto se considera buena práctica almacenar las plantillas que utilizarán los ViewResolver en este directorio.
- El directorio webapp se consideraría la raíz del proyecto, por lo tanto, suponiendo que el archivo se encuentra en la carpeta views/ dentro de WEB-INF/ sólo hay que devolver "/WEB-INF/views/home.jsp".
- Con Spring Boot, se deben indicar el prefijo y el sufijo inyectados automáticamente y se puede utilizar un archivo de configuración src/main/resources/application.properties para especificarlos:

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

JavaServer Pages (JSP)

- JavaServer Pages (JSP) es una de muchas tecnologías que se pueden utilizar para renderizar una vista a partir de modelos de Spring.
- JSP permite generar páginas web dinámicas basadas en documentos estándar de páginas web como HTML o XML.
- Dado que se está usando el patrón MVC hay que limitar el uso de JSP a la presentación y la lógica exclusiva de la presentación.
- Al ser código Java, se trata de código compilado, por lo tanto ofrece ventajas de estabilidad y rapidez frente a alternativas de código interpretado.
- Como desventajas tiene la creación de servlet anémicos y un sistema de etiquetado que dificulta su uso con herramientas de diseño.
- Para trabajar con JSP y JSLT, Spring suministra las “Biblioteca de etiquetas JSP de Spring” y la “Biblioteca de etiquetas de formulario de Spring”.
- Dado que su uso a decaído últimamente, Spring Boot no da soporte por defecto a JSP y requiere su instalación explicita, tanto para JSP como para JSLT.

Configuración JSP

- En el fichero pom.xml:

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

- Crear directorio:
  - /src/main/webapp/WEB-INF/vistas
- Editar src/main/resources/application.properties:

```
spring.mvc.view.prefix=/WEB-INF/vistas/
spring.mvc.view.suffix=.jsp
```

Elementos de JSP

- Una página JSP está hecho de «una plantilla de página», que consiste en:
  - Código HTML,
  - Comentarios JSP y
  - Elementos JSP (Directivas, Secuencias de Comandos (Scriptlets) y Acciones)
- Estos elementos JSP nos permiten insertar código en el servlet que se generará desde la página JSP.
  - Comentarios
  - Directivas
  - Scripting
  - Acciones
- Todos los elementos de JSP comienzan con <% y finalizan con %>
- Para poder introducir un comentario en una página JSP utilizaremos la siguiente sintaxis: <%-- comentario --%>

Directivas

- Las directivas le permiten controlar la estructura general del contenedor JSP (al Servidor JEE) cuyo propósito es proporcionar información para la traducción.
- No depende de la petición ejecutada y no devuelve ningún tipo de respuesta.
  - <%@ directiva attribute1="value1“ … attributeN="valueN" %>
- Las directivas mas comunes son:
  - PAGE <%@ page Atributo="Valor" %>
  - TAGLIB <%@ taglib Atributo="Valor" %>
  - INCLUDE <%@ include file="....." %>

Directiva PAGE

- Permite definir los atributos globales para toda la página (importación de clases, personalizar la superclase del servlet, establecer el tipo de contenido, etc).
- Una directiva de la página aunque se puede colocar en cualquier lugar dentro del documento suele ir al principio.

```
<%@ page language="java" contentType="text/html; charset=UTF-8"
    pageEncoding="UTF-8"%>
```

Atributos	Descripción
buffer = Tamaño	Especifica el tamaño del buffer que será utilizado por el objeto OUT
contentType = "text/html"	Especifica el tipo de archivo que se devuelve
errorPage = "URL"	Pagina de envío para cuando el JSP de una excepción.
import = "paquete | clase"	Importación de un paquete o clase determinada <%@ page import="java.util.Date" %>
languaje = "java"	Define el lenguaje de programación de los SCRIPTING. Por defecto es JAVA

Directiva INCLUDE

- Se utiliza para introducir ficheros estáticos dentro de una página HTML
- El fichero puede tener cualquier contenido de tipos texto incluido código JSP (*.html, *.txt, *.jsp)
- Si introducimos código HTML debe respetar el anidamiento de etiqueta, no podemos tener dos <html>
- Una directiva include se debe colocar en el documento en el punto en el que desea que el archivo que se inserte.

```
<%@ include file="../parts/header.jsp" %>
...
<%@ include file="../parts/footer.jsp" %>
```

Directiva TAGLIB

- Las etiquetas JSP utilizan la sintaxis XML, por lo que se integran limpia y uniformemente a las etiquetas HTML.
- La directiva taglib, introducida en JSP 1.1, indica al motor jsp que la página va a utilizar “tag libraries” o librerías de etiquetas .

  <%@ taglib uri=“ . . . . . ” prefix = “ . . . . . ” %>
  - URI permite localizar el fichero descriptor de la librería de extensión tld.
  - PREFIX especifica el identificador que todas las etiquetas de la librería deben incorporar
- Estas librerías contienen etiquetas creadas por el propio programador con sus correspondientes atributos que encapsulan determinada funcionalidad.
- Estas librerías suelen estar añadidas a los proyecto dentro de META-INF
- Suelen ir al principio de la página después de la directiva PAGE

  <%@taglib uri="http://java.sun.com/jsp/jstl/core" prefix="c"%>
  <%@taglib uri="http://www.springframework.org/tags" prefix="s"%>
  <%@taglib uri="http://www.springframework.org/tags/form" prefix="sf" %>

JSTL

- JSP Standard Tag Library es un conjunto de librerías de etiquetas simples y estándares que encapsulan la funcionalidad principal que es usada comúnmente para escribir páginas JSP.
- Las etiquetas JSTL están organizadas en librerías:
  - core: Comprende las funciones script básicas como loops, condicionales, y entrada/salida.
  - xml: Comprende el procesamiento de xml
  - formatting: Comprende la internacionalización y formato de valores como de moneda y fechas.
  - sql: Comprende el acceso a base de datos.
  - functions: Comprende funciones comunes de manipulación de cadenas
- Las librerías incluyen la mayoría de funcionalidad que será necesaria en una página JSP. Las etiquetas JSTL son muy sencillas de usar por personas que no conocen programación y solo cuentan con conocimientos de etiquetas del estilo HTML.
- Al usar JSLT, se debe usar la clase de vista especial JstlView, ya que, como JSTL necesita algo de preparación antes de que funcionen cosas como las características I18N.

Instalación y configuración del JSTL

- La librería JSTL es distribuida como un conjunto de archivos JAR que simplemente tenemos que agregar en el classpath del contenedor de servlets.
- Debemos usar un contenedor de servlets compatible con la versión JSP 2.0 para usar el JSTL 1.1.
- Descargar la implementación JSTL de la página de proyecto Jakarta TagLibs [http://jakarta.apache.org/taglibs/binarydist.html].
  - Copiar todos los archivos JAR que se encuentran en jakarta-taglibs/standard-1.0/lib al directorio /WEB-INF/lib de la aplicación Web.
- Importar en las páginas JSP las librerías JSTL necesarias mediante las directivas taglib apropiadas:
  - <%@ taglib prefix="c" uri="http://java.sun.com/jsp/jstl/core" %>
  - <%@ taglib prefix="x" uri="http://java.sun.com/jsp/jstl/xml" %>
  - <%@ taglib prefix="fmt" uri="http://java.sun.com/jsp/jstl/fmt" %>
  - <%@ taglib prefix="sql" uri="http://java.sun.com/jsp/jstl/sql" %>
  - <%@ taglib prefix="fn" uri="http://java.sun.com/jsp/jstl/functions" %>

jstl/core

<c:out >	Al igual que <% = ...>, pero las expresiones adecuadas.
<c:set >	Establece el resultado de la evaluación de la expresión en un ámbito.
<c:remove >	Elimina una variable del ámbito (o de un ámbito concreto, si se especifica).
<c:catch>	Atrapa cualquier excepción que se produce en su cuerpo y, opcionalmente, la expone.
<c:if>	Condicional simple ejecuta su cuerpo si la condición es verdadera.
<c:choose>	Código condicional múltiple que establece un contexto para las operaciones condicionales que se excluyen mutuamente, marcadas por <c:when> y <c:otherwise>.
<c:when>	Subetiqueta de <c:choose> que incluye su cuerpo si su expresión es verdadera.
<c:otherwise>	Subetiqueta de <c:choose> que sigue a las etiquetas <c:when> y se ejecuta sólo si todas las condiciones anteriores son falsas
<c:import>	Recupera una URL absoluta o relativa y expone su contenido en la página.
<c:forEach >	Recorre una colección.
<c:forTokens>	Recorre los tokens suministrados, separados por delimitadores.
<c:param>	Añade un parámetro a la URL que contiene una etiqueta de "importación".
<c:redirect >	Redirige a una nueva URL.

jstl/core

```
<c:if test="${user.visitCount == 1}">
This is your first visit. Welcome to the site!
</c:if>

<c:choose>
    <c:when test="${count == 0}”>No records matched your selection.</c:when>
    <c:otherwise>
    <c:out value="${count}"/> records matched your selection.
    </c:otherwise>
</c:choose>

<table>
    <c:forEach var=”customer” items=”${customers}”>
    <tr><td><c:out value=”${customer}”/></td></tr>
    </c:forEach>
</table>
```


jstl/fmt

<fmt:formatNumber>	Para hacer que el valor numérico con precisión o formato específico.
<fmt:parseNumber>	Analiza la representación de cadena de un número, moneda, o porcentaje.
<fmt:formatDate>	Formatea una fecha y / u hora utilizando los estilos y patrones suministrados
<fmt:parseDate>	Analiza la representación de cadena de una fecha y / u hora
<fmt:bundle>	Carga un paquete de recursos para ser utilizados por el cuerpo de la etiqueta.
<fmt:setLocale>	Almacena la configuración regional dada en la variable de configuración local.
<fmt:setBundle>	Carga un paquete de recursos y lo almacena en la variable llamada de ámbito o la variable de configuración paquete.
<fmt:timeZone>	Especifica la zona horaria para la acción de formato de tiempo o de análisis anidados en su cuerpo.
<fmt:setTimeZone>	Almacena la zona horaria dada en la variable de configuración de zona horaria
<fmt:message>	Para mostrar un mensaje de internacionalización.
<fmt:requestEncoding>	Establece la codificación de caracteres de solicitud

jstl/fmt

```
<p>
    <c:set var="now" value="<%=new java.util.Date()%>" />
    Fecha:<br><fmt:formatDate type = "date" value = "${now}" /><br>
    F.Personalizada:<br><fmt:formatDate pattern="dd/MM/yyyy" value="${now}"
        /><br>
    Hora:<br><fmt:formatDate type="time" value="${now}" /><br>
    Fecha y hora:<br><fmt:formatDate type="both" dateStyle = "short"
        timeStyle="short" value="${now}" />
    <c:set var = "value" value = "120000.2309" />
    Decimales:<br><fmt:formatNumber type="number" maxFractionDigits = "2" value
        = "${value}" /><br>
    N.Personalizado:<br><fmt:formatNumber type = "number" pattern = "###.###E0"
        value = "${value}" /><br>
    Porcetaje:<br><fmt:formatNumber type = "percent"  maxIntegerDigits="2"
        minFractionDigits="2" value = "${value}" /><br>
    Moneda:<br><fmt:formatNumber type = "currency" value = "${value}"/>
</p>
```


jstl/xml

<x:out>	Igual que <%= ...>, pero para las expresiones XPath.
<x:parse>	Se utiliza para analizar los datos XML especificados ya sea a través de un atributo o en el cuerpo de la etiqueta.
<x:set>	Establece una variable para el valor de una expresión XPath.
<x:if>	Evalúa una expresión XPath, si es cierta se procesa de su cuerpo pero si es falsa, el cuerpo se ignora.
<x:forEach>	Para recorrer los nodos de un documento XML.
<x:choose>	Código condicional múltiple que establece un contexto para las operaciones condicionales que se excluyen mutuamente, marcadas por <x:when> y <x:otherwise>.
<x:when>	Subetiqueta de <x:choose> que incluye su cuerpo si su expresión es verdadera.
<x:otherwise>	Subetiqueta de <x:choose> que sigue a las etiquetas <x:when> y se ejecuta sólo si todas las condiciones anteriores son falsas
<x:transform>	Se aplica una transformación XSL en un documento XML
<x:param>	Utilizada junto con la etiqueta de transformación para establecer un parámetro en la hoja de estilo XSLT

jstl/xml

```
<!-- parse an XML document -->
<c:import url="http://acme.com/customer?id=76567" var="xml"/>
<x:parse xml="${xml}" var="doc"/>
<!-- access XML data via XPath expressions -->
<x:out select="$doc/name"/>
<x:out select="$doc/address"/>
<!-- set a scoped variable -->
<x:set var="custName" scope="request" select="$doc/name"/>

<!-- context set by ancestor tag <x:forEach> -->
<x:forEach select="$doc//customer">
<x:out select="name"/>
</x:forEach>
```


jstl/functions

fn:contains()	Comprueba si una cadena de entrada contiene la sub-cadena especificada.
fn:containsIgnoreCase()	Comprueba si una cadena de entrada contiene la sub-cadena especificada de una manera insensible a minúsculas y mayúsculas.
fn:endsWith()	Comprueba si una cadena de entrada termina con el sufijo especificado.
fn:escapeXml()	Escapa los caracteres que podrían ser interpretados como marcado XML.
fn:indexOf()	Devuelve el índice de la primera aparición de una sub-cadena especificada dentro de una cadena.
fn:join()	Une todos los elementos de una matriz en una cadena.
fn:length()	Devuelve el número de elementos en una colección o el número de caracteres de una cadena.

jstl/functions

fn:replace()	Devuelve una cadena resultante de la sustitución en una cadena de entrada de todas las ocurrencias con una cadena dada.
fn:split()	Divide una cadena en una matriz de sub-cadenas.
fn:startsWith()	Comprueba si una cadena de entrada comienza con el prefijo especificado.
fn:substring()	Devuelve una sub-cadena de una cadena.
fn:substringAfter()	Devuelve una sub-cadena de una cadena después de una sub-cadena específica.
fn:substringBefore()	Devuelve una sub-cadena de una cadena antes de una sub-cadena específica.
fn:toLowerCase()	Convierte todos los caracteres de una cadena a minúsculas.
fn:toUpperCase()	Convierte todos los caracteres de una cadena a mayúsculas.
fn:trim()	Elimina los espacios en blanco de ambos extremos de una cadena.

EL (Expression Language) en JSP

- EL (Expression Language) es un lenguaje utilizado en las paginas JSP para interactuar con los datos (JavaBeans) servidos (Servlet) por parte del servidor, sin importar el alcance de los atributos (request, session, application). Combinado con la librería JSTL Core (JavaServer Pages Standard Tag Library) nos permite construir toda la lógica de las páginas JSP de una forma mucho mas versátil.
- EL fue agregada a partir de la tecnología JSP 2.0.
  - Para utilizarse no es necesario hacer ninguna declaración especial en los JSP. El único requisito es que sea un archivo JSP válido (e.j. page.jsp).
  - Con EL no es posible modificar los atributos de los Javabeans (métodos setters) únicamente se pueden leer sus atributos (métodos getters).
  - EL permite utilizar objetos implícitos (objetos que no necesitan declararse) y operadores.
  - Con EL no es posible iterar objetos como Arrays, Lists, etc. Para esto, es necesario combinar EL con JSTL.
- Para obtener el valor de cualquier variable, sea del tipo que sea incluidas las propiedades de los objetos (como si fueran atributos) y los elementos de los arrays, lo único que tenemos que hacer es escribir su nombre entre ${}.
  - ${expr}
- La expresiones pueden utilizar operadores aritméticos, lógicos y de comparación.
- Permite el uso de funciones si van precedidas por su taglib prefix.

Objetos Implícitos

- Los objetos implícitos permiten acceder al contexto haciendo innecesario traspasar su información a través del modelo desde el controlador.
- El objeto pageContext le da acceso al objeto JSP pageContext. A través del objeto pageContext, puede acceder al objeto de solicitud.

  ${pageContext.request.queryString}

  ${pageContext.request.contextPath}
- Los objetos header y headerValues le dan acceso a los valores de encabezado normalmente
  - ${header.user-agent}
  - ${header["user-agent"]} .
  - ${header.accept-language}

Objetos Implícitos

Objeto	Descripción
pageScope	Variables en el ámbito del alcance de la página
requestScope	Variables de ámbito de alcance de solicitud
sessionScope	Variables con alcance del alcance de la sesión
aplicaciónScope	Variables en el ámbito de aplicación
param	Solicitar parámetros como cadenas
paramValues	Solicitar parámetros como colecciones de cadenas
header	Encabezados de solicitud HTTP como cadenas
headerValues	Encabezados de solicitud HTTP como colecciones de cadenas
initParam	Parámetros de inicialización de contexto
cookie	Valores de cookie
pageContext	El objeto JSP PageContext para la página actual

Bibliotecas Spring

Biblioteca de etiquetas JSP de Spring

- Spring proporciona el enlace de datos de los parámetros de solicitud a los objetos de comando. Para facilitar el desarrollo de páginas JSP en combinación con estas características de enlace de datos, Spring proporciona algunas etiquetas que facilitan aún más las cosas. Todas las etiquetas Spring tienen características de escape HTML para habilitar o inhabilitar el escape de caracteres.
- El descriptor spring.tld de la biblioteca de etiquetas (TLD) se incluye en el spring-webmvc.jar.

Biblioteca de etiquetas de formulario de primavera

- A partir de la versión 2.0, Spring proporciona un conjunto completo de etiquetas que reconocen el enlace de datos para el manejo de elementos de formulario al usar JSP y Spring Web MVC. Cada etiqueta proporciona soporte para el conjunto de atributos de su contraparte en la etiqueta HTML correspondiente, haciendo que las etiquetas sean familiares e intuitivas de usar. Las etiquetas HTML generadas son compatibles con HTML 4.01 y XHTML 1.0.
- A diferencia de otras bibliotecas de etiquetas de formulario/entrada, la biblioteca de etiquetas de formulario de Spring se integra con Spring Web MVC, dando a las etiquetas acceso al objeto de comando y a los datos de referencia del controlador, haciendo que los JSP sean más fáciles de desarrollar, leer y mantener.

Biblioteca de etiquetas JSP de Spring

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

- <s:bind>
  - Exporta un estado de propiedad enlazada a una propiedad de estado de ámbito de página. Se utiliza junto con <s:path> para obtener un valor de propiedad enlazada.
- <s:escapeBody>
  - Escapa el contenido HTML y/o JavaScript en el cuerpo de la etiqueta.
- <s:hasBindErrors>
  - Representa el contenido de forma condicional si un objeto modelo específico (en un atributo de solicitud) tiene errores de enlace.
- <s:htmlEscape>
  - Establece el valor de escape HTML como predeterminado para la página actual.
- <s:message>
  - Recupera el mensaje con el código dado y lo procesa (predeterminado) o lo asigna a una variable de página, solicitud, sesión o aplicación (cuando se usan los atributos var y scope).

Biblioteca de etiquetas JSP de Spring

- <s:nestedPath>
  - Establece una ruta anidada para ser utilizada por <s: bind>.
- <s:theme>
  - Recupera un mensaje de tema con el código dado y lo procesa (predeterminado) o lo asigna a una variable de página, solicitud, sesión o aplicación (cuando se usan los atributos var y scope).
- <s:transform>
  - Transforma las propiedades no contenidas en un objeto de comando usando los editores de propiedades de un objeto de comando.
- <s:url>
  - Crea direcciones URL relativas al contexto con soporte para variables de plantilla URI y HTML / XML / JavaScript escaping. Puede representar la URL (predeterminado) o asignarla a una variable de página, solicitud, sesión o aplicación (cuando se usan los atributos var y scope).
- <s:eval>
  - Evalúa las expresiones de Spring Expression Language (SpEL), representa el resultado (predeterminado) o lo asigna a una variable de página, solicitud, sesión o aplicación (cuando se usan los atributos var y scope).

Listado

```
@GetMapping("/empleados")
  public String list(Model model, @PageableDefault(size=10, sort = {"firstName",
        "lastName"}) Pageable pageable, HttpServletRequest request) {
    Page<Employee> rslt = dao.findAll(pageable);
     model.addAttribute("listado", rslt);
     model.addAttribute("rslt", rslt);
     return "employees/list";
  }
  @GetMapping("/empleados/{id}")
  public String get(@PathVariable("id") Long id, Model model) {
    Optional<Employee> item = dao.findById(id);
    if (!item.isPresent())
        throw new NotFoundException();
     model.addAttribute("elemento",item.get());
     return "employees/view";
  }
```

Listado

```
<h1>Lista de Empleados</h1>
<table class="table table-hover table-striped table-bordered">
	<tr>
		<th>Empleados</th>
		<td><a href="${pageContext.request.contextPath}/empleados/add" class="btn btn-success">
			<i class="fas fa-plus"></i></a></td>
	</tr>
	<c:forEach var="elemento" items="${listado.getContent()}">
		<tr>
			<td><a href="${pageContext.request.contextPath}/empleados/${elemento.employeeId}">
				${ elemento.firstName } ${ elemento.lastName } </a></td>
			<td>
				<a href="${pageContext.request.contextPath}/empleados/${elemento.employeeId}/edit"
					class="btn btn-success"><i class="fas fa-pen"></i></a>
				<a href="${pageContext.request.contextPath}/empleados/${elemento.employeeId}/delete"
					class="btn btn-success"><i class="fas fa-trash"></i></a>
			</td>
		</tr>
	</c:forEach>
</table>
```


Paginación

```
<ul class="pagination">
	<c:if test = "${listado.hasPrevious()}">
	  <li class="page-item">
		<a class="page-link" href="${pageContext.request.contextPath}/empleados?page=${listado.getNumber()-
			1}">
			<span aria-hidden="true">&laquo;</span>
		</a>
	  </li>
	</c:if>
	<c:forEach var = "i" begin = "0" end = "${listado.getTotalPages() -1}">
	<li class="page-item<c:if test = "${i==listado.getNumber()}"> active</c:if>">
		<a class="page-link" href="${pageContext.request.contextPath}/empleados?page=${i}">${i + 1}</a>
	</li>
	</c:forEach>
	<c:if test = "${listado.hasNext()}">
	  <li class="page-item">
		<a class="page-link"
			href="${pageContext.request.contextPath}/empleados?page=${listado.getNumber()+1}">
			<span aria-hidden="true">&raquo;</span>
		</a>
	  </li>
	</c:if>
</ul>
```

Biblioteca de etiquetas de formulario de Spring

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

<sf:form>	Representa una etiqueta HTML <form> y una ruta de enlace expuesta a las etiquetas internas para el enlace de datos.
<sf:errors>	Representa los errores de campo en una etiqueta HTML .
<sf:label>	Representa una etiqueta HTML <label>.
<sf:input>	Representa una etiqueta HTML <input> con el tipo establecido en texto.
<sf:password>	Representa una etiqueta HTML <entrada> con el tipo establecido en contraseña.
<sf:checkbox>	Muestra una etiqueta HTML <input> con el tipo establecido en checkbox.
<sf:checkboxes>	Representa múltiples etiquetas HTML <input> con el tipo establecido en checkbox.
<sf:radiobutton>	Representa una etiqueta HTML <input> con el tipo establecido en radio.
<sf:radiobuttons>	Representa varias etiquetas HTML <input> con el tipo establecido en radio.
<sf:hidden>	Representa una etiqueta HTML <input> con el tipo establecido en oculto.
<sf:select>	Representa una etiqueta HTML <select>.
<sf:option>	Representa una etiqueta <option> HTML. El atributo seleccionado se establece de acuerdo con el valor límite.
<sf:options>	Muestra una lista de etiquetas <option> de HTML correspondientes a la colección, matriz o mapa enlazado.
<sf:textarea>	Representa una etiqueta HTML <textarea>.

Formularios: Añadir

```
@GetMapping("/add")
public String add(Model model) {
    model.addAttribute("action", "regiones/add");
    model.addAttribute("elemento", new Region());
    return "region/form";
}
@PostMapping("/add")
public ModelAndView addPost(@ModelAttribute("elemento") @Valid Region item, BindingResult
    result) {
    ModelAndView mv = new ModelAndView();
    if (result.hasErrors()) {
        mv.addObject("action", "regiones/add");
        mv.addObject("elemento", item);
        mv.setViewName("region/form");
    } else {
        dao.save(item);
        mv.setViewName("redirect:/regiones");
    }
    return mv;
}
```

Formularios: Modificar

```
@GetMapping("/{id}/edit")
public String edit(@PathVariable("id") Long id, Model model) {
    Optional<Region> item = dao.findById(id);
    if (!item.isPresent())
        throw new NotFoundException();
    model.addAttribute("action", "regiones/" + id + "/edit");
    model.addAttribute("elemento", item.get());
    return "region/form";
}
@PostMapping("/{id}/edit")
public String editPost(@PathVariable("id") Long id, @ModelAttribute("elemento") @Valid Region item, BindingResult
    result, Model model) {
    if(id != item.getRegionId())
        result.addError(new FieldError("elemento", "regionId", "No coincide la clave"));
    if (result.hasErrors()) {
        model.addAttribute("action", "regiones/" + id + "/edit");
        model.addAttribute("elemento", item);
        return "region/form";
    } else {
        dao.save(item);
        return "redirect:/regiones";
    }
}
```


Formularios: Borrar

```
@GetMapping("/{id}/delete")
public String delete(@PathVariable("id") Long id, Model model) {
    try {
        dao.deleteById(id);
    } catch (Exception e) {
    }
    return "redirect:/regiones";
}
```

Formularios

```
<sf:form modelAttribute="elemento" action="${pageContext.request.contextPath}/${action}">
    <sf:errors path="*" cssClass="alert alert-danger" element="div"/>
    <div class="form-group">
        <sf:label path="regionId">ID</sf:label>
        <sf:input path="regionId" cssClass="form-control" cssErrorClass="is-invalid form-control" />
        <sf:errors path="regionId" cssClass="invalid-feedback" />
    </div>
    <div class="form-group">
        <sf:label path="regionName">Nombre</sf:label>
        <sf:input path="regionName" cssClass="form-control"
            cssErrorClass="is-invalid form-control"/>
        <sf:errors path="regionName" cssClass="invalid-feedback" />
    </div>
    <div class="form-group">
        <input class="btn btn-primary" type="submit" value="Enviar"/>
        <a href="${pageContext.request.contextPath}/regiones" class="btn btn-primary">Volver</a>
    </div>
</sf:form>
```


Formatos numéricos y fecha/hora

- Una fuente inagotable de conflictos son los diferentes formatos numéricos y de fecha/hora.
- Spring 3.0 introdujo las anotaciones @DateTimeFormat y @NumberFormat como parte de Formatter SPI para analizar e imprimir valores de campo localizados en aplicaciones web.
- @DateTimeFormat
  - Acepta el formatos basado en ISO (DateTimeFormat.ISO) o basados en un patrón: "dd/MM/yyyy hh:mm:ss".
  - Se puede establecer el estilo con 2 caracteres, el primero para la fecha y el segundo para la hora: 'S' para estilo corto, 'M' para medio, 'L' para largo y 'F' para completo. Se puede omitir una fecha o una hora especificando el carácter de estilo '-'.
- @NumberFormat
  - Acepta el formatos basado en estilo (DEFAULT, NUMBER, PERCENT, CURRENCY) o basados en un patrón: "#.###,##".

Validación

- Es necesario agregar en Spring Boot la referencia Validation.
- Anotar los atributos de la clase del modelo con las restricciones de los valores permitidos (JSR 303).
- Alternativamente se puede crear un validador personalizado.
- Anotar con @Valid el parámetro de entrada del modelo y asociarlo con @ModelAttribute a la propiedad vinculada al formulario, añadir a continuación (adjunto) un parámetro BindingResult que recogerá los errores en la validación y supervisar mediante el método hasErrors() si se han producido errores.

```
addPost(@ModelAttribute("elemento") @Valid Region item, BindingResult result)
```

- Mostrar en el formulario los errores de validación asociados a los controles de entrada del formulario:

```
<sf:errors path="regionId" />
```

- Alternativamente se puede mostrar un sumario con todos los errores producidos en el formulario:

```
<sf:errors path="*" cssClass="alert alert-danger" element="div"/>
```


Validación personalizada

- Una alternativa diferente a validar con anotaciones en el modelo es crear nuestra propia lógica de validación para un formulario en especifico, para utilizar esta funcionalidad necesitamos la anotación @InitBinder y además implementar la interface Validator.
- La interface Validator solo tiene dos métodos:
  - supports: indica las clases a la que el validador le da soporte.
  - validate: realiza la validación y, si hay errores, registra los correspondientes mensajes.
- La clase ValidationUtils da soporte a la validación y el registro de errores.
- Un método anotado con @InitBinder se encargara de la tarea de registrar el validador en el WebDataBinder.

Validación personalizada

```
public class UsuarioValidator implements Validator {
	@Override
	public boolean supports(Class<?> clazz) {
		return true; //Usuario.class.equals(clazz);
	}
	@Override
	public void validate(Object target, Errors errors) {
		if(target instanceof Usuario) {
			Usuario item = (Usuario) target;
		  ValidationUtils.rejectIfEmptyOrWhitespace(errors, "nombre", "usuario.nombre", "Es obligatorio.");
		  ValidationUtils.rejectIfEmptyOrWhitespace(errors, "password", "usuario.password", "Es obligatorio.");
		  if(!item.getPassword().equals(item.getPassword2()))
			errors.rejectValue("password2", "usuario.password2", "No coincide");
		}
	}

}

@InitBinder
protected void initBinder(WebDataBinder binder) {
   binder.setValidator(new UsuarioController.UsuarioValidator());
}
```


Internacionalización (I18n)

- Las aplicaciones basadas en Spring Framework soportan I18N, es decir podremos mostrar los textos o mensajes de la aplicación en diversos idiomas.
- Los mensajes se almacenan en archivos de propiedades (pares clave/valor), uno por cada idioma disponible en la aplicación, el nombre de este archivo debe tener el siguiente formato: messages_<language_code>_<country_code>.properties donde <language_code> debe ser el código de lenguaje y <country_code> el código de región que es opcional.
  - message_en.properties
    title: Hello I18N with Spring
    text: Bye {0}
  - message_es.properties
    title: Hola I18N con Spring
    text: Adios {0}

Internacionalización (I18n)

- Para utilizar esta funcionalidad se debe agregar un bean de tipo ResourceBundleMessageSource en la configuración:

```
@Bean
  public ResourceBundleMessageSource messageSource() {
    ResourceBundleMessageSource rbms = new
ResourceBundleMessageSource();
    rbms.setBasename("i18n/messages"); // Directorio de los ficheros
    return rbms;
  }
```

- Inyectando MessageSource se puede acceder a los mensajes i18n a través del método getMessage().

```
@Autowired private MessageSource ms;

String msg_en = ms.getMessage("title", null, Locale.ENGLISH);
String msg_es = ms.getMessage("title", null, new Locale("es"));
String txt_en = ms.getMessage("text", new Object[]{name}, Locale.ENGLISH);
String txt_es = ms.getMessage("text", new Object[]{name}, new Locale("es"));
```


Vistas en múltiples idiomas

- Adicionalmente, para poder utilizar I18n en las vistas, hay que configurar en una clase que implemente WebMvcConfigurer una serie de bean.
- Un bean LocaleResolver para establecer el lugar en donde se almacenará la información del idioma que utiliza el usuario.
- Un bean SessionLocaleResolver para guardar esta información en la sesión (existen más implementaciones como CookieLocaleResolver que guarda la información en una cookie)
- Un bean LocaleChangeInterceptor que intercepta la petición HTTP y cambia el idioma actual si la URL contiene el parámetro lang=idioma, donde idioma es el código del lenguaje deseado: en, es, fr, uk, etc., el nombre del parámetro se establece con el método setParamName("..."), por ejemplo “lang”.
  - Para cambiar a español: url?lang=es
  - Para cambiar a inglés: url?lang=en
- Es necesario registrar el interceptor sobre escribiendo el método addInterceptors de la interfaz WebMvcConfigurer.
- Los textos en los JSP deben utilizar el Spring Tag message

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

Configuración múltidioma

```
@Bean
public MessageSource messageSource() {
   ResourceBundleMessageSource messageSource = new ResourceBundleMessageSource();
   messageSource.setBasename("i18n/messages");
   return messageSource;
}
@Bean
public LocaleResolver localeResolver() {
   return new SessionLocaleResolver();
}
@Bean
public LocaleChangeInterceptor localeChangeInterceptor() {
   LocaleChangeInterceptor localeChangeInterceptor = new LocaleChangeInterceptor();
   localeChangeInterceptor.setParamName("lang");
   return localeChangeInterceptor;
}
@Override
public void addInterceptors(InterceptorRegistry registry) {
   registry.addInterceptor(localeChangeInterceptor());
}
```


SERVICIOS REST

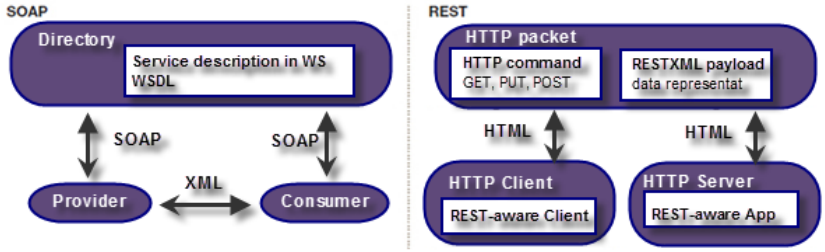
REST (REpresentational State Transfer)

- Un **estilo de arquitectura** para desarrollar aplicaciones web distribuidas que se basa en el uso del protocolo HTTP e Hypermedia.
- Definido en el 2000 por Roy Fielding, para no reinventar la rueda, se basa en aprovechar lo que ya estaba definido en el HTTP pero que no se utilizaba.
- El HTTP ya define 8 métodos (algunas veces referidos como "verbos") que indica la acción que desea que se efectúe sobre el recurso identificado:
  - HEAD, GET, POST, PUT, DELETE, TRACE, OPTIONS, CONNECT
- El HTTP permite en el encabezado transmitir la información de comportamiento:
  - Accept, Content-type, Response (códigos de estado), Authorization, Cache-control, …

Introducción a los Servicios REST

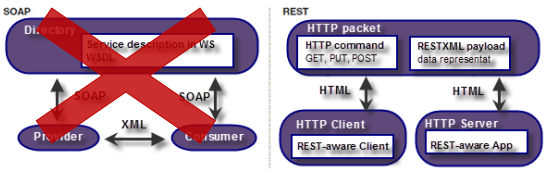
- REST y SOAP son muy diferentes.
  - SOAP:
    - Es un protocolo de mensajería
  - REST:
    - Es un estilo de arquitectura de software para sistemas hipermedia distribuidos.
    - Sistemas en los que el texto, gráficos, audio y otros medios de comunicación se almacenan en una red e interconectados a través de hipervínculos .
- WWW es un sistema REST (la web estática, la web dinámica no).
  - En la Web, HTTP es a la vez un protocolo de transporte y un sistema de mensajería (las peticiones y respuestas HTTP son mensajes).
- Estos requisitos REST son:
  - Se publican Recursos (Un dato, una operación, un numero de empleado, el empleado 44, etc.)
  - Los servicios REST no publican un conjunto de métodos u operaciones, publican RECURSOS.
  - Cada recurso dispone de un identificador único.
  - Cada recurso debe de tener una o varias representaciones de su estado (XML, HTML, PDF, etc)

Introducción a los Servicios REST



REST (REpresentation State Transfer)

- Nos permite utilizar cualquier interfaz web simple que utiliza HTTP, sin las abstracciones/restricciones de los protocolos basados en patrones de intercambio de mensajes.
- Los servicios que siguen los principios de REST habitualmente se denominan RESTful.
- REST se basa en varios componentes principales:
  - Recursos
  - URI
  - Representaciones
  - Solicitudes HTTP

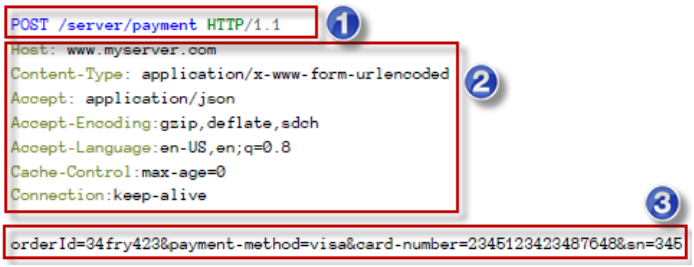


HTTP

- HTTP es una pieza fundamental en World Wide Web, y especifica como intercambiar entre cliente y servidor recursos web.
- Es un protocolo idóneo para implementar servicios web, ya que sigue los principios REST.
- Características de HTTP:
  - Es un protocolo de nivel de aplicación y algo de presentación.
  - Está diseñado para ser ejecutado sobre TCP o sobre TLS/SSL.
  - Se basa en un paradigma sencillo de petición/respuesta, es decir, es un protocolo stateless.

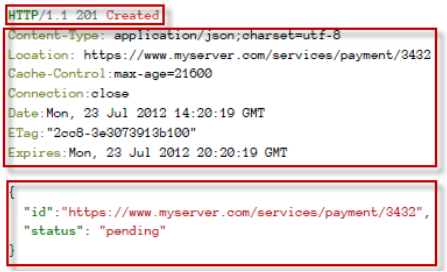
Petición HTTP

- Cuando realizamos una petición HTTP, el mensaje consta de:
  - Primera línea de texto indicando la versión del protocolo utilizado, el verbo y el URI
    - El verbo indica la acción a realizar sobre el recurso web localizado en la URI
  - Posteriormente vendrían las cabeceras (opcionales)
  - Después el cuerpo del mensaje, que contiene un documento, que puede estar en cualquier formato (XML, HTML, JSON → Content-type)



Respuesta HTTP

- Los mensajes HTTP de respuesta siguen el mismo formato que los de envío.
- Sólo difieren en la primera línea
  - Donde se indica un código de respuesta junto a una explicación textual de dicha respuesta.
  - El código de respuesta indica si la petición tuvo éxito o no.



Recursos

- Un recurso es cualquier elemento que dispone de un URI correcto y único.
- Es cualquier cosa que sea direccionable a través de internet.
- Estos recursos pueden ser manipulados por clientes y servidores.
  - Una noticia.
  - La temperatura en Madrid a las 22:00h.
  - Un estudiante de alguna clase en alguna escuela
  - Un ejemplar de un periódico, etc
- En REST todos los recursos comparten una interfaz única y constante. (http://...)
- Todos los recursos tienen las mismas operaciones (CRUD)
  - CREATE, READ, UPDATE, DELETE

URI (Uniform Resource Identifier)

- Los URI son los identificadores globales de recursos en la web, y actúan de manera efectiva como UUIDs REST.
- Hay 2 tipos de URIs : URL y URN
  - URLs Identifican un recurso de red mediante una IP o un DNS
  - URNs son simples UUIDs lógicos con un espacio de nombres asociados
- URI es una cadena de caracteres corta, que identifica inequívocamente un recurso y que tienen el siguiente formato

  <esquema>://<host>:puerto/<ruta><querystring><fragmento>
  - Esquema = Indican que protocolo hay que utilizar para usar el recurso (http o https)
  - Host = Indica el lugar donde encontraremos el recurso (por IP o por dominio)
  - Puerto = Puerto por donde se establece la conexión (80 o 443)
  - Ruta = Ruta del recurso dentro del servidor, está separado por /
  - queryStrng = Parámetros adicionales, separados por ? o por &
  - Fragmento = Separado por #

URI (Uniform Resource Identifier)

- Las URI es el único medio por el que los clientes y servidores pueden realizar el intercambio de representaciones.
- Normalmente estos recursos son accesibles en una red o sistema.
- Para que un URI sea correcto, debe de cumplir los requisitos de formato, REST no indica de forma específica un formato obligatorio.
- Los URI asociados a los recursos pueden cambiar si modificamos el recurso (nombre, ubicación, características, etc)

Métodos HTTP

HTTP	REST	Descripción
GET	RETRIEVE	Sin identificador: Recuperar el estado completo de un recurso (HEAD + BODY) Con identificador: Recuperar el estado individual de un recurso (HEAD + BODY)
HEAD		Recuperar el estado de un recurso (HEAD)
POST	CREATE or REPLACE	Crea o modifica un recurso (sin identificador)
PUT	CREATE or REPLACE	Crea o modifica un recurso (con identificador)
DELETE	DELETE	Sin identificador: Elimina todo el recurso Con identificador: Elimina un elemento concreto del recurso
CONNECT		Comprueba el acceso al host
TRACE		Solicita al servidor que introduzca en la respuesta todos los datos que reciba en el mensaje de petición
OPTIONS		Devuelve los métodos HTTP que el servidor soporta para un URL específico
PATCH	REPLACE	HTTP 1.1 Reemplaza parcialmente un elemento del recurso

Tipos MIME

- Otro aspecto muy importante es la posibilidad de negociar distintos formatos (representaciones) a usar en la transferencia del estado entre servidor y cliente (y viceversa).
- La representación de los recursos es el formato de lo que se envía un lado a otro entre clientes y servidores.
- Como REST utiliza HTTP, podemos transferir múltiples tipos de información.
- Los datos se transmiten a través de TCP/IP, el navegador sabe cómo interpretar las secuencias binarias (Content-Type) por el protocolo HTTP
- La representación de un recurso depende del tipo de llamada que se ha generado (Texto, HTML, PDF, etc).
- En HTTP cada uno de estos formatos dispone de su propio tipos MIME, en el formato <tipo>/<subtipo>.
  - application/json application/xml text/html text/plain image/jpeg

Tipos MIME

- Para negociar el formato entre el cliente y el servidor se utilizan las cabeceras:
  - Petición
    - En la cabecera ACCEPT se envía una lista de tipos MIME que el cliente entiende.
    - En caso de enviar contenido en el cuerpo, la cabecera CONTENT-TYPE indica en que formato MIME está codificado.
  - Respuesta
    - El servidor selecciona el tipo que más le interese de entre todos los especificados en la cabecera ACCEPT, y devuelve la respuesta indicando con la cabecera CONTENT-TYPE el formato del cuerpo.
- La lista de tipos MIME se especifica en la cabecera (ACCEPT) mediante lo que se llama una lista separada por comas de tipos (media range). También pueden aparecer expresiones de rango, por ejemplo
  - */* indica cualquier tipo MIME
  - image / * indica cualquier formato de imagen
- Si el servidor no entiende ninguno de los tipos MIME propuestos (ACCEPT) devuelve un mensaje con código 406 (incapaz de aceptar petición).

Códigos HTTP (status)

status	statusText	Descripción
100	Continue	Una parte de la petición (normalmente la primera) se ha recibido sin problemas y se puede enviar el resto de la petición
101	Switching protocols	El servidor va a cambiar el protocolo con el que se envía la información de la respuesta. En la cabecera Upgrade indica el nuevo protocolo
200	OK	La petición se ha recibido correctamente y se está enviando la respuesta. Este código es con mucha diferencia el que mas devuelven los servidores
201	Created	Se ha creado un nuevo recurso (por ejemplo una página web o un archivo) como parte de la respuesta
202	Accepted	La petición se ha recibido correctamente y se va a responder, pero no de forma inmediata
203	Non-Authoritative Information	La respuesta que se envía la ha generado un servidor externo. A efectos prácticos, es muy parecido al código 200
204	No Content	La petición se ha recibido de forma correcta pero no es necesaria una respuesta
205	Reset Content	El servidor solicita al navegador que inicialice el documento desde el que se realizó la petición, como por ejemplo un formulario
206	Partial Content	La respuesta contiene sólo la parte concreta del documento que se ha solicitado en la petición

Códigos de redirección

status	statusText	Descripción
300	Multiple Choices	El contenido original ha cambiado de sitio y se devuelve una lista con varias direcciones alternativas en las que se puede encontrar el contenido
301	Moved Permanently	El contenido original ha cambiado de sitio y el servidor devuelve la nueva URL del contenido. La próxima vez que solicite el contenido, el navegador utiliza la nueva URL
302	Found	El contenido original ha cambiado de sitio de forma temporal. El servidor devuelve la nueva URL, pero el navegador debe seguir utilizando la URL original en las próximas peticiones
303	See Other	El contenido solicitado se puede obtener en la URL alternativa devuelta por el servidor. Este código no implica que el contenido original ha cambiado de sitio
304	Not Modified	Normalmente, el navegador guarda en su caché los contenidos accedidos frecuentemente. Cuando el navegador solicita esos contenidos, incluye la condición de que no hayan cambiado desde la última vez que los recibió. Si el contenido no ha cambiado, el servidor devuelve este código para indicar que la respuesta sería la misma que la última vez
305	Use Proxy	El recurso solicitado sólo se puede obtener a través de un proxy, cuyos datos se incluyen en la respuesta
307	Temporary Redirect	Se trata de un código muy similar al 302, ya que indica que el recurso solicitado se encuentra de forma temporal en otra URL

Códigos de error del navegador

status	statusText	Descripción
400	Bad Request	El servidor no entiende la petición porque no ha sido creada de forma correcta
401	Unauthorized	El recurso solicitado requiere autorización previa
402	Payment Required	Código reservado para su uso futuro
403	Forbidden	No se puede acceder al recurso solicitado por falta de permisos o porque el usuario y contraseña indicados no son correctos
404	Not Found	El recurso solicitado no se encuentra en la URL indicada. Se trata de uno de los códigos más utilizados y responsable de los típicos errores de *Página no encontrada*
405	Method Not Allowed	El servidor no permite el uso del método utilizado por la petición, por ejemplo por utilizar el método GET cuando el servidor sólo permite el método POST
406	Not Acceptable	El tipo de contenido solicitado por el navegador no se encuentra entre la lista de tipos de contenidos que admite, por lo que no se envía en la respuesta
407	Proxy Authentication Required	Similar al código 401, indica que el navegador debe obtener autorización del proxy antes de que se le pueda enviar el contenido solicitado
408	Request Timeout	El navegador ha tardado demasiado tiempo en realizar la petición, por lo que el servidor la descarta

Códigos de error del navegador

status	statusText	Descripción
409	Conflict	El navegador no puede procesar la petición, ya que implica realizar una operación no permitida (como por ejemplo crear, modificar o borrar un archivo)
410	Gone	Similar al código 404. Indica que el recurso solicitado ha cambiado para siempre su localización, pero no se proporciona su nueva URL
411	Length Required	El servidor no procesa la petición porque no se ha indicado de forma explícita el tamaño del contenido de la petición
412	Precondition Failed	No se cumple una de las condiciones bajo las que se realizó la petición
413	Request Entity Too Large	La petición incluye más datos de los que el servidor es capaz de procesar. Normalmente este error se produce cuando se adjunta en la petición un archivo con un tamaño demasiado grande
414	Request-URI Too Long	La URL de la petición es demasiado grande, como cuando se incluyen más de 512 bytes en una petición realizada con el método GET
415	Unsupported Media Type	Al menos una parte de la petición incluye un formato que el servidor no es capaz procesar
416	Requested Range Not Suitable	El trozo de documento solicitado no está disponible, como por ejemplo cuando se solicitan bytes que están por encima del tamaño total del contenido
417	Expectation Failed	El servidor no puede procesar la petición porque al menos uno de los valores incluidos en la cabecera Expect no se pueden cumplir

Códigos de error del servidor

status	statusText	Descripción
500	Internal Server Error	Se ha producido algún error en el servidor que impide procesar la petición
501	Not Implemented	Procesar la respuesta requiere ciertas características no soportadas por el servidor
502	Bad Gateway	El servidor está actuando de proxy entre el navegador y un servidor externo del que ha obtenido una respuesta no válida
503	Service Unavailable	El servidor está sobrecargado de peticiones y no puede procesar la petición realizada
504	Gateway Timeout	El servidor está actuando de proxy entre el navegador y un servidor externo que ha tardado demasiado tiempo en responder
505	HTTP Version Not Supported	El servidor no es capaz de procesar la versión HTTP utilizada en la petición. La respuesta indica las versiones de HTTP que soporta el servidor

Uso de la cabecera

- Request: Método /uri?parámetros
  - GET: Recupera el recurso
    - Todos: Sin parámetros
    - Uno: Con parámetros
  - POST: Crea un nuevo recurso
  - PUT: Edita el recurso
  - DELETE: Elimina el recurso
- Accept: Indica al servidor el formato o posibles formatos esperados, utilizando MIME.
- Content-type: Indica en que formato está codificado el cuerpo, utilizando MIME
- Response: Código de estado con el que el servidor informa del resultado de la petición.

Objetivos de los servicios REST

- Desacoplar el cliente del backend
- Mayor escalabilidad
  - Sin estado en el backend.
- Separación de problemas
- División de responsabilidades
- API uniforme para todos los clientes
  - Disponer de una interfaz uniforme (basada en URIs)

Diseño de un Servicio Web REST

- Para el desarrollo de los Servicios Web's REST es necesario conocer una serie de cosas:
  - Analizar el/los recurso/s a implementar
  - Diseñar la REPRESENTACION del recurso.
    - Deberemos definir el formato de trabajo del recurso: XML, JSON, HTML, imagen, RSS, etc
  - Definir el URI de acceso.
    - Deberemos indicar el/los URI de acceso para el recurso
  - Conocer los métodos soportados por el servicio
    - GET, POST, PUT, DELETE
  - Qué códigos de estado pueden ser devueltos
    - Los códigos de estado HTTP típicos que podrían ser devueltos
- Todo lo anterior dependerá del servicio a implementar.

Peticiones

- Request: GET /users
- Response: 200
  - content-type:application/json
  - BODY
- Request: GET /users/11
- Response: 200
  - content-type:application/json
  - BODY
- Request: POST /users
  - BODY
- Response: 201
  - content-type:application/json
  - BODY
- Request: PUT /users
  - BODY
- Response: 200
  - content-type:application/json
  - BODY
- Request: DELETE /users/11
- Response: 204 no content

Richardson Maturity Model

http://www.crummy.com/writing/speaking/2008-QCon/act3.html

- # Nivel 1 (Pobre): Se usan URIs para identificar recursos:
  - Se debe identificar un recurso
    /invoices/?page=2 → /invoices/page/2
  - Se construyen con nombres nunca con verbos
    /getUser/{id} → /users/{id}/
    /users/{id}/edit/login → users/{id}/access-token
  - Deberían tener una estructura jerárquica
    /invoices/user/{id} → /user/{id}/invoices
- # Nivel 2 (Medio): Se usa el protocolo HTTP adecuadamente
- # Nivel 3 (Óptimo): Se implementa hypermedia.

Hypermedia

- Se basa en la idea de enlazar recursos: propiedades que son enlaces a otros recursos.
- Para que sea útil, el cliente debe saber que en la respuesta hay contenido hypermedia.
- En content-type es clave para esto
  - Un tipo genérico no aporta nada:
    Content-Type: text/xml
  - Se pueden crear tipos propios
    Content-Type:application/servicio+xml

JSON Hypertext Application Language

- RFC4627 http://tools.ietf.org/html/draft-kelly-json-hal-00
- HATEOAS: Content-Type: application/hal+json

```
{
    "_links": {
      "self": {"href": "/orders/523" },
      "warehouse": {"href": "/warehouse/56" },
      "invoice": {"href": "/invoices/873“}
    },
    , "currency": "USD"
    , "status": "shipped"
    , "total": 10.20
}
```


Recursos

- Son clases Java con la anotación @RestController (@Controller y @ResponseBody) y representan a los servicios REST, son controller que reciben y responden a las peticiones de los clientes.

```
@RestController()
public class PaisController {
        @Autowired
        private PaisRepository paisRepository;
```

- Los métodos de la clase que interactúan con el cliente deben llevar la anotación @RequestMapping, con la subruta y el RequestMethod.

```
@RequestMapping(value = "/paises/{id}", method = RequestMethod.GET)
public ResponseEntity<Pais> getToDoById(@PathVariable("id") String id) {
    return new ResponseEntity<Pais>(paisRepository.findById(id).get(),
            HttpStatus.OK);
}
```

Respuesta

- La anotación @ResponseBody en un método indica que el retorno será serializado en el cuerpo de la respuesta a través de un HttpMessageConverter.

```
@PostMapping("/invierte")
@ResponseBody
public Punto body(@RequestBody Punto p) {
    int x = p.getX();
    p.setX(p.getY());
    p.setY(x);
    return p;
}
```

- El código de estado de la respuesta se puede establecer con la anotación @ResponseStatus:

```
@PostMapping
@ResponseStatus(HttpStatus.CREATED)
public void add(@RequestBody Punto p) { … }
```


Respuesta personalizada

- La clase ResponseEntity permite agregar estado y encabezados a la respuesta (no requiere la anotación @ResponseBody).

```
@GetMapping(value="/pais")
public ResponseEntity<List<Pais>> getAll(){
    return new ResponseEntity<List<Pais>>(
            paisRepository.findAll(),
            HttpStatus.OK);
}
```

- La clase ResponseEntity dispone de builder para generar la respuesta:

```
return ResponseEntity.ok().eTag(etag).build(body);
```

Paginación y Ordenación

QueryString	Descripción
page	Número de página en base 0. Por defecto: página 0.
size	Tamaño de página. Por defecto: 20.
sort	Propiedades de ordenación en el formato property,property(,ASC|DESC). Por defecto: ascendente. Hay que utilizar varios sort para diferente direcciones (?sort=firstname&sort=lastname,asc)

```
@GetMapping
public List<Employee> getAll(Pageable pageable) {
        if(pageable.isPaged()) {
                return dao.findAll(pageable).getContent();
        } else
                return dao.findAll();
}
```


Mapeo de respuestas genéricas a excepciones.

- Un requisito común para los servicios REST es incluir detalles de error en el cuerpo de la respuesta.
- Spring Framework no lo hace automáticamente porque la representación de los detalles de error en el cuerpo de la respuesta es específica de la aplicación.
- Una clase @RestController puede contar con métodos anotados con @ExceptionHandler que intercepten determinadas excepciones producidas en el resto de los métodos de la clase y devuelven un ResponseEntity que permite establecer el estado y el cuerpo de la respuesta.
- Esto mismo se puede hacer globalmente en clases anotadas con @ControllerAdvice que solo tienen los correspondientes métodos @ExceptionHandler.

Excepciones personalizadas

```
public class NotFoundException extends Exception {
    private static final long serialVersionUID = 1L;
    public NotFoundException() {
        super("NOT FOUND");
    }
    public NotFoundException(String message) {
        super(message);
    }
    public NotFoundException(Throwable cause) {
        super("NOT FOUND", cause);
    }
    public NotFoundException(String message, Throwable cause) {
        super(message, cause);
    }
    public NotFoundException(String message, Throwable cause, boolean enableSuppression, boolean
        writableStackTrace) {
        super(message, cause, enableSuppression, writableStackTrace);
    }
}
```


Error Personalizado

```
public class ErrorMessage implements Serializable {
    private static final long serialVersionUID = 1L;
    private String error, message;
    public ErrorMessage(String error, String message) {
        this.error = error;
        this.message = message;
    }
    public String getError() { return error; }
    public void setError(String error) { this.error = error; }
    public String getMessage() { return message; }
    public void setMessage(String message) { this.message = message; }
}
```

@ControllerAdvice

```
@ControllerAdvice
public class ApiExceptionHandler {
   @ResponseStatus(HttpStatus.NOT_FOUND)
   @ExceptionHandler({NotFoundException.class})
   @ResponseBody
   public ErrorMessage notFoundRequest(HttpServletRequest request, Exception exception) {
      return new ErrorMessage(exception.getMessage(), request.getRequestURI());
   }

   @ResponseStatus(HttpStatus.BAD_REQUEST)
   @ExceptionHandler({ BadRequestException.class, MalformedHeaderException.class,
         FieldInvalidException.class
   })
   @ResponseBody
   public ErrorMessage badRequest(Exception exception) {
      return new ErrorMessage(exception.getMessage(), "");
   }
}
```


Servicio Web RESTful

```
import javax.validation.ConstraintViolation;
import javax.validation.Valid;
import javax.validation.Validator;
import javax.websocket.server.PathParam;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.http.MediaType;
import org.springframework.http.ResponseEntity;
import org.springframework.web.bind.annotation.DeleteMapping;
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.PathVariable;
import org.springframework.web.bind.annotation.PostMapping;
import org.springframework.web.bind.annotation.PutMapping;
import org.springframework.web.bind.annotation.RequestBody;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.ResponseStatus;
import org.springframework.web.bind.annotation.RestController;
import org.springframework.web.servlet.support.ServletUriComponentsBuilder;
import org.springframework.http.HttpStatus;

import curso.api.exceptions.BadRequestException;
import curso.api.exceptions.NotFoundException;
import curso.model.Actor;
import curso.repositories.ActorRepository;
```

Servicio Web RESTful

```
@RestController
@RequestMapping("/api/actores")
public class ActorResource {
	@Autowired
	private ActorRepository dao;

	@Autowired
	private Validator validator;

	@GetMapping
	public List<Actor> getAll() {
		// …
	}

	@GetMapping(path = "/{id}")
	public Actor getOne(@PathVariable int id) throws NotFoundException {
		// …
	}
```


Servicio Web RESTful

```
	@PostMapping
	public ResponseEntity<Object> create(@Valid @RequestBody Actor item) throws BadRequestException {
		// …
		URI location = ServletUriComponentsBuilder.fromCurrentRequest().path("/{id}")
			.buildAndExpand(newItem.getActorId()).toUri();
		return ResponseEntity.created(location).build();

	}

	@PutMapping("/{id}")
	@ResponseStatus(HttpStatus.ACCEPTED)
	public void update(@PathVariable int id, @Valid @RequestBody Actor item) throws BadRequestException,
			NotFoundException {
		// …
	}

	@DeleteMapping("/{id}")
	@ResponseStatus(HttpStatus.NO_CONTENT)
	public void delete(@PathVariable int id) {
		// ..
	}
}
```

Spring Data Rest

- Spring Data REST se basa en los repositorios de Spring Data y los exporta automáticamente como recursos REST. Aprovecha la hipermedia para que los clientes encuentren automáticamente la funcionalidad expuesta por los repositorios e integren estos recursos en la funcionalidad relacionada basada en hipermedia.
- Spring Data REST es en sí misma una aplicación Spring MVC y está diseñada de tal manera que puede integrarse con las aplicaciones Spring MVC existentes con un mínimo esfuerzo.
- De forma predeterminada, Spring Data REST ofrece los recursos REST en la URI raíz, '/', se puede cambiar la URI base configurando en el fichero application.properties:
  - spring.data.rest.basePath=/api
- Dado que la funcionalidad principal de Spring Data REST es exportar como recursos los repositorios de Spring Data, el artefacto principal será la interfaz del repositorio.

Spring Data Rest

- Spring Data REST es expone los metodos del repositorio como métodos REST:
  - GET: findAll(), findAll(Pageable), findAll(Sort)
    - Si el repositorio tiene capacidades de paginación, el recurso toma los siguientes parámetros:
      - page: El número de página a acceder (base 0, por defecto a 0).
      - size: El tamaño de página solicitado (por defecto a 20).
      - sort: Una colección de directivas de género en el formato ($propertyname,)+[asc|desc]?.
  - POST, PUT, PATCH: save(item)
  - DELETE: delete(id)
- Devuelve el conjunto de códigos de estado predeterminados:
  - 200 OK: Para peticiones GET .
  - 201 Created: Para solicitudes POST y PUT que crean nuevos recursos.
  - 204 No Content: Para solicitudes PUT, PATCH y DELETE cuando está configurada para no devolver cuerpos de respuesta para actualizaciones de recursos (RepositoryRestConfiguration.returnBodyOnUpdate). Si se configura incluir respuestas para PUT, se devuelve 200 OK para las actualizaciones y 201 Created si crea nuevos recursos.

Spring Data Rest

- Para cambiar la configuración predeterminada del REST:

```
@RepositoryRestResource(path="personas", rel="persona",
collectionResourceRel="personas")
public interface PersonaRepository extends JpaRepository<Persona, Integer> {
    @RestResource(path = "por-nombre")
    List<Person> findByNombre(String nombre);
    // http://localhost:8080/personas/search/nombre?nombre=terry
```

- Para ocultar ciertos repositorios, métodos de consulta o campos

```
@RepositoryRestResource(exported = false)
interface PersonaRepository extends JpaRepository<Persona, Integer> {
    @RestResource(exported = false)
    List<Person> findByName(String name);
    @Override
    @RestResource(exported = false)
    void delete(Long id);
```


Spring Data Rest

- Spring Data REST presenta una vista predeterminada del modelo de dominio que exporta. Sin embargo, a veces, es posible que deba modificar la vista de ese modelo por varias razones.

  Mediante un interfaz en el paquete de las entidades o en uno de subpaquetes se crea un proyección con nombre:

```
@Projection(name = "personasAcortado", types = { Country.class })
public interface PersonaProjection {
    public int getPersonaId();
    public String getNombre();
    public String getApellidos();
}
```

- Para acceder a la proyección:
  - http://localhost:8080/personas?projection=personasAcortado
- Para fijar la proyección por defecto:

```
@RepositoryRestResource(excerptProjection = PersonaProjection.class)
public interface PersonaRepository extends JpaRepository<Persona, Integer> {
```

Spring Data Rest

- Spring Data REST usa HAL para representar las respuestas, que define los enlaces que se incluirán en cada propiedad del documento devuelto.
- Spring Data REST proporciona un documento ALPS (Semántica de perfil de nivel de aplicación) para cada repositorio exportado que se puede usar como un perfil para explicar la semántica de la aplicación en un documento con un tipo de medio agnóstico de la aplicación (como HTML, HAL, Collection + JSON, Siren, etc.).
  - http://localhost:8080/profile
  - http://localhost:8080/profile/personas

Seguridad

- La ejecución de aplicaciones JavaScript puede suponer un riesgo para el usuario que permite su ejecución.
- Por este motivo, los navegadores restringen la ejecución de todo código JavaScript a un entorno de ejecución limitado.
- Las aplicaciones JavaScript no pueden establecer conexiones de red con dominios distintos al dominio en el que se aloja la aplicación JavaScript.
- Los navegadores emplean un método estricto para diferenciar entre dos dominios ya que no permiten ni subdominios ni otros protocolos ni otros puertos.
- Si el código JavaScript se descarga desde la siguiente URL: http://www.ejemplo.com
- Las funciones y métodos incluidos en ese código no pueden acceder a:
  - https://www.ejemplo.com/scripts/codigo2.js
  - http://www.ejemplo.com:8080/scripts/codigo2.js
  - http://scripts.ejemplo.com/codigo2.js
  - http://192.168.0.1/scripts/codigo2.js

CORS

- Un recurso hace una solicitud HTTP de origen cruzado cuando solicita otro recurso de un dominio distinto al que pertenece.
- XMLHttpRequest sigue la política de mismo-origen, por lo que, una aplicación usando XHR solo puede hacer solicitudes HTTP a su propio dominio. Para mejorar las aplicaciones web, los desarrolladores pidieron que se permitieran a XHR realizar solicitudes de dominio cruzado.
- El Grupo de Trabajo de Aplicaciones Web del W3C recomienda el nuevo mecanismo de Intercambio de Recursos de Origen Cruzado (CORS, Cross-origin resource sharing: https://www.w3.org/TR/cors). Los servidores deben indicar al navegador mediante cabeceras si aceptan peticiones cruzadas y con que características:
  - "Access-Control-Allow-Origin", "*"
  - "Access-Control-Allow-Headers", "Origin, Content-Type, Accept, Authorization, X-XSRF-TOKEN"
  - "Access-Control-Allow-Methods", "GET, POST, PUT, DELETE, OPTIONS"
  - "Access-Control-Allow-Credentials", "true"
- Soporte: Chrome 3+ Firefox 3.5+ Opera 12+ Safari 4+ Internet Explorer 8+

CORS

- Para configurar CORS en la interfaz del repositorio

```
@CrossOrigin(origins = "http://myDomain.com", maxAge = 3600,
      methods={RequestMethod.GET, RequestMethod.POST })
public interface PersonaRepository extends JpaRepository<Persona, Integer> {
```

- Para configurar CORS globalmente

```
@Configuration @EnableWebMvc
public class WebConfig implements WebMvcConfigurer {
   @Override
   public void addCorsMappings(CorsRegistry registry) {
      registry.addMapping("/api/**")
         .allowedOrigins("*")
         .allowedMethods("GET", "POST", "PUT", "DELETE")
         .allowedHeaders("origin", "content-type", "accept", "authorization")
         .allowCredentials(true).maxAge(3600);
   }
}
```

CLIENTES DE LOS SERVICIOS REST

RestTemplate

- La RestTemplate proporciona un API de nivel superior sobre las bibliotecas de cliente HTTP y facilita la invocación de los endspoint REST en una sola línea. Para incorporarlo en Maven:

```
<dependency>
   <groupId>org.springframework</groupId>
   <artifactId>spring-web</artifactId>
</dependency>
<dependency>
   <groupId>com.fasterxml.jackson.core</groupId>
   <artifactId>jackson-databind</artifactId>
</dependency>
```

- Para poder inyectar la dependencia:

```
@Bean public RestTemplate restTemplate(RestTemplateBuilder builder) {
      return builder.build();
}
@Autowired RestTemplate srvRest;
```

RestTemplate

Grupo de métodos	Descripción
getForObject	Recupera una representación a través de GET.
getForEntity	Recupera un ResponseEntity(es decir, estado, encabezados y cuerpo) utilizando GET.
headForHeaders	Recupera todos los encabezados de un recurso utilizando HEAD.
postForLocation	Crea un nuevo recurso utilizando POST y devuelve el encabezado Location de la respuesta.
postForObject	Crea un nuevo recurso utilizando POST y devuelve la representación del objeto de la respuesta.
postForEntity	Crea un nuevo recurso utilizando POST y devuelve la representación de la respuesta.
put	Crea o actualiza un recurso utilizando PUT.

RestTemplate

Grupo de métodos	Descripción
patchForObject	Actualiza un recurso utilizando PATCH y devuelve la representación de la respuesta.
delete	Elimina los recursos en el URI especificado utilizando DELETE.
optionsForAllow	Recupera los métodos HTTP permitidos para un recurso utilizando ALLOW.
exchange	Versión más generalizada (y menos crítica) de los métodos anteriores que proporciona flexibilidad adicional cuando es necesario. Acepta a RequestEntity (incluido el método HTTP, URL, encabezados y cuerpo como entrada) y devuelve un ResponseEntity.
execute	La forma más generalizada de realizar una solicitud, con control total sobre la preparación de la solicitud y la extracción de respuesta a través de interfaces de devolución de llamada.

RestTemplate

- Para recuperar uno:

```
PersonaDTO rslt = srvRest.getForObject(
      "http://localhost:8080/api/personas/{id}",
      PersonaDTO.class, 1);
```

- Para recuperar todos (si no se dispone de una implementación de List<PersonaDTO>):

```
ResponseEntity<List<PersonaDTO>> response =
    srvRest.exchange("http://localhost:8080/api/personas",
    HttpMethod.GET,
    HttpEntity.EMPTY, new
    ParameterizedTypeReference<List<PersonaDTO>>() {
    });
List<PersonaDTO> rslt = response.getBody();
```


RestTemplate

- Para crear o modificar un recurso:

```
ResponseEntity<PersonaDTO> httpRslt =
    srvRest.postForEntity(
    "http://localhost:8080/api/personas", new
    PersonaDTO("pepito", "grillo")), PersonaDTO.class);
```

- Para crear o modificar un recurso con identificador:

```
srvRest.put("http://localhost:8080/api/personas/{id}", new
    PersonaDTO(new PersonaDTO("Pepito", "Grillo"))), 111);
```

- Para borrar un recurso con identificador:

```
srvRest.delete("http://localhost:8080/api/personas/{id}",
    111);
```

RestTemplate

- De forma predeterminada, RestTemplate lanzará una de estas excepciones en caso de un error de HTTP:
  - HttpClientErrorException: en estados HTTP 4xx
  - HttpServerErrorException: en estados HTTP 5xx
  - UnknownHttpStatusCodeException: en caso de un estado HTTP desconocido.
- Para vigilar las excepciones:

```
} catch (HttpClientErrorException e) {
    switch (e.getStatusCode()) {
    case BAD_REQUEST:
    case NOT_FOUND:
        // …
        break;
```


LinkDiscoverers

- Cuando se trabaja con representaciones habilitadas para hipermedia, una tarea común es encontrar un enlace con un tipo de relación particular en ellas.
- Spring HATEOAS proporciona implementaciones basadas en JSONPath de la interfaz LinkDiscoverer.

```
<dependency>
   <groupId>com.jayway.jsonpath</groupId>
   <artifactId>json-path</artifactId>
</dependency>
```

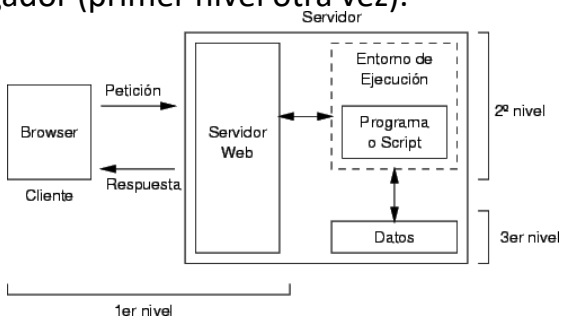
- Para acceder a un enlace:

```
String resp = srvRest.getForObject("http://localhost:8080/personas/1",
      String.class);
LinkDiscoverer discoverer = new HalLinkDiscoverer();
Link link = discoverer.findLinkWithRel("direcciones", resp);
if(link != null)
      direccionesURL = link.getHref();
```

ASYNCHRONOUS JAVASCRIPT AND XML

Introducción

- Una aplicación Web típica recogerá datos del usuario (primer nivel), los enviará al servidor, que ejecutará un programa (segundo y tercer nivel) y cuyo resultado será formateado y presentado al usuario en el navegador (primer nivel otra vez).



Introducción

- AJAX, acrónimo de Asynchronous JavaScript And XML (JavaScript asíncrono y XML), es una técnica de desarrollo web para crear aplicaciones interactivas o RIA (Rich Internet Applications). Estas aplicaciones se ejecutan en el cliente, es decir, en el navegador de los usuarios mientras se mantiene la comunicación asíncrona con el servidor en segundo plano. De esta forma es posible realizar cambios sobre las páginas sin necesidad de recargarlas, mejorando la interactividad, velocidad y usabilidad en las aplicaciones.
- Ajax es una tecnología asíncrona, en el sentido de que los datos adicionales se solicitan al servidor y se cargan en segundo plano sin interferir con la visualización ni el comportamiento de la página. Ajax no constituye una tecnología en sí, sino que es un término que engloba a un grupo de éstas que trabajan conjuntamente.
- JavaScript es el lenguaje interpretado (scripting language) en el que normalmente se efectúan las funciones de llamada de Ajax mientras que el acceso a los datos se realiza mediante XMLHttpRequest, objeto disponible en los navegadores actuales. En cualquier caso, no es necesario que el contenido asíncrono esté formateado en XML.
- Ajax es una técnica válida para múltiples plataformas y utilizable en muchos sistemas operativos y navegadores dado que está basado en estándares abiertos como JavaScript y Document Object Model (DOM).

XMLHttpRequest

- XMLHttpRequest (XHR), también conocido como XMLHTTP (Extensible Markup Language / Hypertext Transfer Protocol), es una interfaz empleada para realizar peticiones HTTP y HTTPS a servidores Web.
- Para los datos transferidos se usa cualquier codificación basada en texto, incluyendo: texto plano, XML, JSON, HTML y codificaciones particulares específicas.
- El navegador implementa la interfaz como una clase de la que una aplicación cliente puede generar tantas instancias como necesite y permita el navegador para manejar el diálogo con el servidor.
- La primera versión de la interfaz XMLHttpRequest fue desarrollada por Microsoft que la introdujo en la versión 5.0 de Internet Explorer utilizando un objeto ActiveX. A partir de la versión 7 la interfaz se ofrece de manera integrada.
- El proyecto Mozilla incorporó la primera implementación integrada en la versión 1.0 de la Suite Mozilla en 2002. Esta implementación sería seguida por Apple a partir de Safari 1.2, Opera Software a partir del Opera 8.0 e iCab desde la versión 3.0b352.
- El World Wide Web Consortium presentó el 27 de septiembre de 2006 el primer borrador de una especificación estándar de la interfaz.

Propiedades

Propiedad	Descripción	
readyState	0	No inicializado (objeto creado, pero no se ha invocado el método open)
	1	Cargando (objeto creado, pero no se ha invocado el método send)
	2	Cargado (se ha invocado el método send, pero el servidor aún no ha respondido)
	3	Interactivo (descargando, se han recibido algunos datos, aunque no se puede emplear la propiedad responseText)
	4	Completo (se han recibido todos los datos de la respuesta del servidor)
responseText	El contenido de la respuesta del servidor en forma de cadena de texto	
responseXML	El contenido de la respuesta del servidor en formato XML. El objeto devuelto se puede procesar como un objeto DOM	
status	El código de estado HTTP devuelto por el servidor (200 para una respuesta correcta, 404 para "No encontrado", 500 para un error de servidor, etc.)	
statusText	El código de estado HTTP devuelto por el servidor en forma de cadena de texto: "OK", "Not Found", "Internal Server Error", etc.	

Métodos y eventos

Método	Descripción
abort()	Detiene la petición actual.
getAllResponseHeaders()	Devuelve una cadena de texto con todas las cabeceras de la respuesta del servidor.
getResponseHeader("cabecera")	Devuelve una cadena de texto con el contenido de la cabecera solicitada.
open("metodo", "url")	Establece los parámetros de la petición que se realiza al servidor. Los parámetros necesarios son el método HTTP empleado y la URL destino.
send(contenido)	Realiza la petición HTTP al servidor
setRequestHeader("cabecera", "valor")	Permite establecer cabeceras personalizadas en la petición HTTP. Se debe invocar entre el open() y el send().
onreadystatechange	**Evento.** Se invoca cada vez que se produce un cambio en el estado de la petición HTTP.

Pasos a seguir

1. Obtener XMLHttpRequest
2. Crear y asignar el controlador del evento onreadystatechange.
3. Abrir conexión: open(method, url*, async*):
4. Opcional. Añadir cabeceras.
5. Opcional. Serializar datos a enviar vía POST.
6. Enviar petición: send()

Obtener XMLHttpRequest

- Los navegadores que siguen los estándares (Firefox, Safari, Opera, Internet Explorer 7+) implementan el objeto XMLHttpRequest de forma nativa, por lo que se puede obtener a través del objeto window. Los navegadores obsoletos (Internet Explorer 5 y 6) implementan el objeto XMLHttpRequest como un objeto de tipo ActiveX.

```
var xmlhttp;
if (window.XMLHttpRequest) {
    //El explorador implementa la interfaz de forma nativa
    xmlhttp = new XMLHttpRequest();
} else if (window.ActiveXObject) {
    //El explorador permite crear objetos ActiveX
    try {
        xmlhttp = new ActiveXObject("MSXML2.XMLHTTP");
    } catch (e) {
        try {
            xmlhttp = new ActiveXObject("Microsoft.XMLHTTP");
        } catch (e) {}
    }
}
if (!xmlhttp) {
    alert("No ha sido posible crear una instancia de XMLHttpRequest");
}
```

Controlador del evento onreadystatechange

- readyState: cuando vale 4 (Completo: se han recibido todos los datos de la respuesta del servidor)
- status: cuando vale 200 (OK: La petición se ha recibido correctamente y se está enviando la respuesta)
- responseText: El contenido de la respuesta del servidor en forma de cadena de texto
- responseXML: El contenido de la respuesta del servidor en formato XML.

```
xmlhttp.onreadystatechange = function () {
   if (xmlhttp.readyState == 4)
      if (xmlhttp.status == 200) {
         xmlDoc = xmlhttp.responseXML;
         // Tratamiento de los datos recivbdos
      } else {
         // Tratamiento de excepción
      }
};
```


Enviar petición

- Abrir conexión: open(method, url, async):
  - method: Verbo HTTP (GET, POST, …)
  - async: Opcional, marcar con false para comportamiento síncrono

```
xmlhttp.open("GET","demo_get.asp?nocache=" + Math.random());
xmlhttp.open("POST","demo_post.asp");
```

- Opcional. Añadir cabeceras.

```
xmlhttp.setRequestHeader("Content-type","application/x-www-form-urlencoded");
```

- Opcional. Serializar datos a enviar vía POST.
  - Reunir los datos a enviar en una cadena formada por pares “nombre=valor” concatenados con ampersand (&).

```
var nombre= document.getElementById("fldNombre");
var datos="nombre="+encodeURIComponent(nombre)
    +"&apellidos="+encodeURIComponent(apellidos);
```

- Enviar petición: send()

```
xmlhttp.send(datos);
```

Texto plano, HTML y JavaScript

- Texto plano

```
var rslt = http_request.responseText;
document.getElementById("myDiv").innerHTML=rslt;
```

- HTML

```
var rslt = http_request.responseText;
document.getElementById("myDiv").innerHTML=rslt;
```

- JavaScript

```
var respuesta = http_request.responseText;
eval(respuesta);
```


XML

- Serializar:

```
var datos="<Datos>";
datos+="<Nombre>"+nombre+"<\/Nombre>";
datos+="<Apellidos>"+apellidos+"<\/Apellidos>";
datos+="<\/Datos>";
```

- Recibir:

```
xmlDoc=xmlhttp.responseXML;
txt="";
x=xmlDoc.getElementsByTagName("Provincias");
for (i=0;i<x.length;i++) {
     txt=txt + x[i].childNodes[0].nodeValue + "<br>";
 }
 document.getElementById("myDiv").innerHTML=txt;
```

JSON

- Serializar:

```
var datos= JSON.stringify(objeto_json);
var datos='{"Datos": {';
datos+='"Nombre":"'+nombre+'"';
datos+=',"Apellidos":"'+apellidos+'"';
datos+='}}';
```

- Recibir:

```
var respuesta = http_request.responseText;
var objeto_json= JSON.parse(respuesta).Provincias;
//var objeto_json = eval("("+respuesta+")").Provincias;
txt="";
for (i=0;i<objeto_json.length;i++) {
  txt=txt + objeto_json[i].Nombre + "<br>";
}
document.getElementById("myDiv").innerHTML=txt;
```


Detener las peticiones

```
…
// Fijar un temporizador que aborte la petición
var temporizador = setTimeout(function() {
   xmlhttp.abort();
   alert(…);
}, 18000); // 2 minutos
xmlhttp.onreadystatechange = function () {
    if (xmlhttp.readyState == 4) {
      clearTimeout(temporizador);
      if (xmlhttp.status == 200) {
         // Eliminar el temporizador, innecesario
…
```

Indicador de descarga

```
…
var temporizador = setTimeout(function() {
   document.getElementById("trabajandoAJAX").style.display="none";
   xmlhttp.abort();
   alert(…);
}, 18000); // 30 segundos
// Muestra el indicador hasta ahora oculto
document.getElementById("trabajandoAJAX").style.display = "block";
xmlhttp.onreadystatechange = function () {
    if (xmlhttp.readyState == 4)
       try {
          if (xmlhttp.status == 200) {
          …
       } finally {
          // Oculta el indicador
          document.getElementById("trabajandoAJAX").style.display="none";
       }
…
```


SEGURIDAD

Spring Security

- Spring Security es un framework de apoyo al marco de trabajo Spring, que dota al mismo de una serie servicios de seguridad aplicables para sistemas basados en la arquitectura JEE, enfocado particularmente sobre proyectos construidos usando Spring Framework. De esta dependencia, se minimiza la curva de aprendizaje si ya se conoce Spring.
- Los procesos de seguridad están destinados principalmente, a comprobar la identidad del usuario mediante la autenticación y los permisos asociados al mismo mediante la autorización. La autorización, basada en roles, es dependiente de la autenticación ya que se produce posteriormente a su proceso.
- Por regla general muchos de estos modelos de autenticación son proporcionados por terceros o son desarrollados por estándares importantes como el IETF. Adicionalmente, Spring Security proporciona su propio conjunto de características de autenticación:
  - In-Memory, JDBC, LDAP, OAuth 2.0, Kerberos, SAML …
- El proceso de autorización se puede establecer a nivel de recurso individual o mediante configuración que cubra múltiples recursos.

Spring Boot

- Si Spring Security está en la ruta de clase, las aplicaciones web están protegidas de forma predeterminada. Spring Boot se basa en la estrategia de negociación de contenido de Spring Security para determinar si se debe usar httpBasic o formLogin.
- Para agregar seguridad a nivel de método a una aplicación web, también puede agregar @EnableGlobalMethodSecurity en la configuración que desee.
- El valor predeterminado del UserDetailsService tiene un solo usuario. El nombre del usuario es “user” y la contraseña se genera aleatoriamente al arrancar y se imprime como INFO:
  - Using generated security password: e4918bc4-d8ac-4179-9916-c37825c7eb55
- Puede cambiar el nombre de usuario y la contraseña proporcionando un spring.security.user.name y spring.security.user.password en application.properties.
- Las características básicas predeterminadas en una aplicación web son:
  - Un bean UserDetailsService con almacenamiento en memoria y un solo usuario con una contraseña generada.
  - Inicio de sesión basado en formularios o seguridad básica HTTP (según el tipo de contenido) para toda la aplicación (incluidos los endpoints).
  - Un DefaultAuthenticationEventPublisher para la publicación de eventos de autenticación.

Seguridad MVC

- La configuración de seguridad predeterminada se implementa en SecurityAutoConfiguration y UserDetailsServiceAutoConfiguration. SecurityAutoConfiguration importa SpringBootWebSecurityConfiguration para la seguridad web y UserDetailsServiceAutoConfiguration configura la autenticación, que también es relevante en aplicaciones no web.
- Para desactivar completamente la configuración de seguridad de la aplicación web predeterminada,se puede agregar un bean de tipo WebSecurityConfigurerAdapter (al hacerlo, no se desactiva la configuración UserDetailsService).
- Para cambiar la configuración del UserDetailsService, se puede añadir un bean de tipo UserDetailsService, AuthenticationProvider o AuthenticationManager.
- Las reglas de acceso se pueden anular agregando una personalización de WebSecurityConfigurerAdapter, que proporciona métodos de conveniencia que se pueden usar para anular las reglas de acceso para los puntos finales del actuador y los recursos estáticos.
- EndpointRequest se puede utilizar para crear un RequestMatcher que se basa en la propiedad management.endpoints.web.base-path. PathRequest se puede usar para crear recursos RequestMatcher en ubicaciones de uso común

Elementos principales

- SecurityContextHolder contiene información sobre el contexto de seguridad actual de la aplicación, que contiene información detallada acerca del usuario que está trabajando actualmente con la aplicación. Utiliza el ThreadLocal para almacenar esta información, que significa que el contexto de seguridad siempre está disponible para la ejecución de los métodos en el mismo hilo de ejecución (Thread). Para cambiar eso, se puede utilizar un método estático SecurityContextHolder.setStrategyName (estrategia de cadena).
- SecurityContext contiene un objeto de autenticación, es decir, la información de seguridad asociada con la sesión del usuario.
- Authentication es, desde punto de vista Spring Security, un usuario (Principal)
- GrantedAuthority representa la autorización dada al usuario de la aplicación.
- UserDetails estandariza la información del usuario independientemente del sistema de autenticación.
- UserDetailsService es la interfaz utilizada para crear el objeto UserDetails.

Proceso de Autenticación

- Para poder tomar decisiones sobre el acceso a los recursos, es necesario que el participante se identifique para realizar las comprobaciones necesarias sobre su identidad. Mediante la interfaz Authentication, se pueden acceder a tres objetos bien diferenciados:
  - principal, normalmente hace referencia al nombre del participante
  - credenciales (del usuario) que permiten comprobar su identidad, normalmente su contraseña, aunque también puede ser otro tipo de métodos como certificados, etc...
  - autorizaciones, un lista de los roles asociados al participante.
- Si un usuario inicia un proceso de autenticación, se crea un objeto Authentication, con los elementos Principal y Credenciales. Si realiza la autenticación mediante el empleo de contraseña y nombre usuario, se crea un objeto UsernamePasswordAuthenticationToken. El framework Spring Security aporta un conjunto de clases que permite que esta autenticación se realice mediante nombre de usuario y contraseña. Para ello, utiliza la autenticación que proporciona el contenedor o utiliza un servicio de identificación basado en Single Sign On (sólo se identifica una vez).

Proceso de Autenticación

- Una vez se ha obtenido el objeto Authentication se envía al AuthenticationManager. Una vez aquí, se realiza una comprobación del contenido de los elementos del objeto principal y las credenciales. Se comprueban que concuerdan con las esperadas, añadiéndole al objeto Authentication las autorizaciones asociadas a esa identidad o generandon una excepción de tipo AuthenticationException.
- El propio framework ya tiene implementado un gestor de autenticación que es válido para la mayoría de los casos, el ProviderManager. El bean AuthenticationManager es del tipo ProviderManager, lo que significa que actúa de proxy con el AuthenticationProvider.
- Este es el encargado de realizar la comprobación de la validez del nombre de usuario/contraseña asociada y de devolver las autorizaciones permitidas a dicho participante (roles asociados).
- Esta clase delega la autenticación en una lista que engloba a los proveedores y que, por tanto, es configurable. Cada uno de los proveedores tiene que implementar el interfaz AuthenticationProvider.

Proceso de Autenticación

- Cada aplicación web tendrá una estrategia de autenticación por defecto. Cada sistema de autenticación tendrá su AuthenticationEntryPoint propio, que realiza acciones como enviar avisos para la autenticación.
- Cuando el navegador decide presentar sus credenciales de autenticación (ya sea como formulario HTTP o HTTP header) tiene que existir algo en el servidor que "recoja" estos datos de autenticación. A este proceso se le denomina "mecanismo de autenticación". Una vez que los detalles de autenticación se recogen en el agente de usuario, un objeto "solicitud de autenticación" se construye y se presenta a un AuthenticationProvider.
- El último paso en el proceso de autenticación de seguridad es un AuthenticationProvider. Es el responsable de tomar un objeto de solicitud de autenticación y decidir si es o no válida. El Provider decide si devolver un objeto de autenticación totalmente lleno o una excepción.
- Cuando el mecanismo de autenticación recibe de nuevo el objeto de autenticación, si se considera la petición válida, debe poner la autenticación en el SecurityContextHolder, y hacer que la solicitud original se ejecute. Si, por el contrario, el AuthenticationProvider rechazó la solicitud, el mecanismo de autenticación mostrara un mensaje de error.

Proceso de Autenticación

- El DaoAuthenticationProvider es una implementación de la interfaz de autenticación centrada en el acceso a los datos que se encuentran almacenados dentro de una base de datos. Este proveedor específico requiere una atención especial.
- Esta implementación delega a su vez en un objeto de tipo UserDetailsService, un interfaz que define un objeto de acceso a datos con un único método loadUserByUsername que permite obtener la información de un usuario a partir de su nombre de usuario devolviendo un UserDetails que estandariza la información del usuario independientemente del sistema de autenticación.
- El UserDetails contiene el nombre de usuario, contraseña, los flags isAccountNonExpired, isAccountNonLocked, isCredentialsNonExpired, IsEnabled y los roles del usuario.
- Los roles de usuario son cadenas que por defecto llevan el prefijo de “ROLE_”.

Cifrado de claves

- Nunca se debe almacenar las contraseñas en texto plano, uno de los procesos básicos de seguridad contra robo de identidad es el cifrado de las claves de usuario.
- Spring Security ofrece algoritmos de encriptación que se pueden aplicar de forma rápida al resto de la aplicación.
- Para esto hay que utilizar una clase que implemente la interfaz PasswordEncoder, que se utilizará para cifrar la contraseña introducida a la hora de crear el usuario.
- Además, hay que pasárselo al AuthenticationManagerBuilder cuando se configura para que cifre la contraseña recibida antes de compararla con la almacenada.
- Spring suministra BCryptPasswordEncoder que es una implementación del algoritmo BCrypt, que genera una hash segura como una cadena de 60 caracteres.

```
@Autowired private PasswordEncoder passwordEncoder;

String encodedPass = passwordEncoder.encode(userDTO.getPassword());
```


Configuración de Autenticación

- Para realizar la configuración crear una clase, anotada con @Configuration y @EnableWebSecurity, que extienda a WebSecurityConfigurerAdapter.
- La sobreescritua del método configure(AuthenticationManagerBuilder) permite fijar el UserDetailsService y el PasswordEncoder.

```
@Configuration
@EnableWebSecurity
@EnableGlobalMethodSecurity(prePostEnabled = true)
public class SecurityConfig extends WebSecurityConfigurerAdapter {
   @Autowired
   UserDetailsService userDetailsService;
   @Bean
   public PasswordEncoder passwordEncoder() {
        return new BCryptPasswordEncoder();
   }
   @Autowired
   public void configure(AuthenticationManagerBuilder auth) throws Exception {
        auth.userDetailsService(userDetailsService)
        .passwordEncoder(passwordEncoder());
   }
}
```

UserDetailsService

```
@Service
@Transactional
public class UserDetailsServiceImpl implements UserDetailsService {
	@Autowired
	private PasswordEncoder passwordEncoder;
	@Override
	public UserDetails loadUserByUsername(final String username) throws UsernameNotFoundException {
		switch(username) {
		case "user":		return this.userBuilder(username, passwordEncoder.encode("user"), "USER");
		case "manager":	return this.userBuilder(username, passwordEncoder.encode("manager"), "MANAGER");
		case "admin":		return this.userBuilder(username, passwordEncoder.encode("admin"), "USER",
                                                "MANAGER", "ADMIN");
		default: throw new UsernameNotFoundException("Usuario no encontrado");
		}
	}
	private User userBuilder(String username, String password, String... roles) {
		List<GrantedAuthority> authorities = new ArrayList<>();
		for (String role : roles) {
			authorities.add(new SimpleGrantedAuthority("ROLE_" + role));
		}
		return new User(username, password, /* enabled */ true, /* accountNonExpired */ true,
				/* credentialsNonExpired */ true, /* accountNonLocked */ true, authorities);
	}}
```


InMemoryAuthentication

```
@Autowired
  public void configureAuth(AuthenticationManagerBuilder auth)
      throws Exception {
   auth.inMemoryAuthentication()
   .withUser("user")
      .password("user").roles("USER")
   .and()
   .withUser("manager")
      .password("manager").roles("MANAGER")
   .and()
   .withUser("admin")
      .password("admin").roles("USER", "ADMIN");
  }
```

Autenticación por formularios

- Spring Security genera automáticamente la URL “/login” y la página de inicio de sesión con el formulario de autenticación.
- Para habilitar la autenticación por formularios y, opcionalmente, cambiar a una página personalizada:

```
protected void configure(HttpSecurity http) throws Exception {
   http.authorizeRequests().and()
      .formLogin()
        .loginPage("/mylogin")
        .permitAll();
}
```

- En una implementación de WebMvcConfigurer se registra la página:

```
public void addViewControllers(ViewControllerRegistry registry) {
      registry.addViewController("/mylogin").setViewName("mylogin");
      registry.setOrder(Ordered.HIGHEST_PRECEDENCE);
}
```

- El formulario debe enviarse vía POST a la misma URL, dode el nombre de usuario se envía en el parámetro HTTP "username" y la contraseña en "password". Si param.error != null es por que el nombre de usuario o la contraseña son inválidos y el param.logout != null se debe a la redirección después de desconectar.

Autenticación por formularios

```
<c:url value="/mylogin" var="loginUrl" />
<form action="${loginUrl}" method="post">
    <c:if test="${param.error != null}">
        <div class="alert alert-danger">Invalid username and password.</div>
    </c:if>
    <c:if test="${param.logout != null}">
        <div class="alert alert-info">You have been logged out.</div>
    </c:if>
    <div class="form-group">
        <label for="username">Username</label>
        <input type="text" id="username" name="username" class="form-control"/>
    </div>
    <div class="form-group">
        <label for="password">Password</label>
        <input type="password" id="password" name="password" class="form-control"/>
    </div>
    <div class="form-group">
        <input type="hidden" name="${_csrf.parameterName}" value="${_csrf.token}" />
        <input type="submit" class="btn btn-primary">
    </div>
</form>
```

Recordar usuario

- Otro concepto básico en la seguridad es la opción de recordar el usuario. Si se recuerda el usuario, la sesión se iniciará automáticamente en el navegador hasta que se cierre sesión voluntariamente (o se borren los datos de sesión asociados). Para conseguir esto es necesario enviar el parámetro HTTP "_spring_security_remember_me" (por defecto) en la petición de inicio de sesión.
- Se configura un bean que implemente la clase RememberMeServices. En este caso se utiliza TokenBasedRememberMeServices, que guarda una cookie codificada en base64 indicando el usuario recordado. Para crear este objeto hace falta una clave privada que se almacena en application.properties para poderla reconfigurar con comodidad.

```
@Value("${rememberMe.privateKey}") private String rememberMeKey;
@Bean    public RememberMeServices rememberMeServices() {
     return new TokenBasedRememberMeServices(rememberMeKey, userDetailsService);
}
```

- En una implementación de WebMvcConfigurer se registra la bean

```
protected void configure(HttpSecurity http) throws Exception {
  http.authorizeRequests().and()
        .rememberMe().key(rememberMeKey)
        .rememberMeServices(rememberMeServices())
```


Autorización

- El AccessDecisionManager es la interfaz que atiende la llamada AbstractSecurityInterceptor producida tras interceptar una petición. Esta interfaz es la responsable final de la toma de decisiones sobre el control de acceso.
- AccessDecisionManager delega la facultad de emitir votos en objetos de tipo AccessDecisionVoter. Se proporcionan dos implementaciones de éste último interfaz:
  - RoleVoter, que comprueba que el usuario presente un determinado rol, comprobando si se encuentra entre sus autorizaciones (authorities).
  - BasicAclEntryVoter, que a su vez delega en una jerarquía de objetos que permite comprobar si el usuario supera las reglas establecidas como listas de control de acceso.
- El acceso por roles se puede fijar para:
  - URLs, permitiendo o denegando completamente
  - Servicios, controladores o métodos individuales

Configuración

- La sobreescritua del método configure(HttpSecurity) permite configurar el http.authorizeRequests():
  .antMatchers("/static/**").permitAll() acceso a los recursos
  .anyRequest().authenticated() se requiere estar autenticado para todas las peticiones.
  .antMatchers("/**").permitAll() equivale a anyRequest()
  .antMatchers("/privado/**", "/config/**").authenticated() equivale a @PreAuthorize("authenticated")
  .antMatchers("/admin/**").hasRole("ADMIN") equivale a @PreAuthorize("hasRole('ROLE_ADMIN')")
- El método .and() permite concatenar varias definiciones.

Seguridad: Configuración

```
@Configuration
@EnableWebSecurity
@EnableGlobalMethodSecurity(prePostEnabled = true)
public class SecurityConfig extends WebSecurityConfigurerAdapter {
   // …
   @Override
   protected void configure(HttpSecurity http) throws Exception {
      http.csrf().disable()
         .authorizeRequests()
            .antMatchers("/**").permitAll()
            .antMatchers("/privado/**").authenticated()
            .antMatchers("/admin/**").hasRole("ADMIN")
         .and()
            .formLogin().loginPage("/login").permitAll()
         .and().logout().permitAll();
   }
}
```

Basada en anotaciones

- Desde la versión 2.0 en adelante, Spring Security ha mejorado sustancialmente el soporte para agregar seguridad a los métodos de capa de servicio proporcionando soporte para la seguridad con anotación JSR-250, así como la anotación original @Secured del marco. A partir de la 3.0 también se puede hacer uso de nuevas anotaciones basadas en expresiones .
- Se puede habilitar la seguridad basada anotaciones utilizando la anotación @EnableGlobalMethodSecurity en cualquier instancia @Configuration.
- @Secured: Anotación para definir una lista de atributos de configuración de seguridad para métodos de un servicio y se puede utilizar como una alternativa a la configuración XML.

```
@Secured({ "ROLE_USER" }) public void create(Contact contact) {
@Secured({ "ROLE_USER", "ROLE_ADMIN" }) public void update(Contact contact) {
@Secured({ "ROLE_ADMIN" }) public void delete(Contact contact){
```

- @PreAuthorize: Anotación para especificar una expresión de control de acceso al método que se evaluará para decidir si se permite o no una invocación del método.

```
@PreAuthorize("isAnonymous()")
@PreAuthorize("hasAuthority('ROLE_TELLER')")
@PreAuthorize("authenticated")
@PreAuthorize("hasRole('USER')")
@PreAuthorize("hasPermission(#contact, 'admin')")
```

- @PostAuthorize: Anotación para especificar una expresión de control de acceso al método que se evaluará después de que se haya invocado un método.

Control de acceso basado en expresiones

Expresión	Descripción
hasRole([role])	Devuelve true si el principal actual tiene el rol especificado. De forma predeterminada, si el rol proporcionado no comienza con 'ROLE_' se agregará. Esto se puede personalizar modificando el defaultRolePrefix en DefaultWebSecurityExpressionHandler.
hasAnyRole([role1,role2])	Se devuelve true si el principal actual tiene alguno de los roles proporcionados (lista de cadenas separadas por comas).
hasAuthority([authority])	Devuelve true si el principal actual tiene la autoridad especificada.
hasAnyAuthority([authority1,authority2])	Se devuelve true si el principal actual tiene alguna de las autorizaciones proporcionadas (se proporciona como una lista de cadenas separadas por comas)
principal	Permite el acceso directo al objeto principal que representa al usuario actual.
authentication	Permite el acceso directo al objeto Authentication actual obtenido del SecurityContext

Control de acceso basado en expresiones

Expresión	Descripción
permitAll	Siempre se evalúa a true
denyAll	Siempre se evalúa a false
isAnonymous()	Devuelve true si el principal actual es un usuario anónimo
isRememberMe()	Devuelve true si el principal actual es un usuario de recordarme
isAuthenticated()	Devuelve true si el usuario no es anónimo
isFullyAuthenticated()	Se devuelve true si el usuario no es un usuario anónimo o recordado
hasPermission(Object target, Object permission)	Devuelve true si el usuario tiene acceso al objetivo proporcionado para el permiso dado. Por ejemplo, hasPermission(domainObject, 'read')
hasPermission(Object targetId, String targetType, Object permission)	Devuelve true si el usuario tiene acceso al objetivo proporcionado para el permiso dado. Por ejemplo, hasPermission(1, 'com.example.domain.Message', 'read')

Cierre de sesión

- Al usar el WebSecurityConfigurerAdapter las opciones de cierre de sesión se establecen automáticamente. Por defecto basta con acceder a la URL /logout y se cerrará la sesión del usuario:
  - Invalidando la sesión HTTP
  - Limpiando cualquier autenticación RememberMe que este configurada
  - Limpiando el SecurityContextHolder
  - Y redirige a /login?logout
- Para personalizar las opciones:

```
protected void configure(HttpSecurity http) throws Exception {
   http
      .logout()
         .logoutUrl("/my/logout")
         .logoutSuccessUrl("/my/index")
         .logoutSuccessHandler(logoutSuccessHandler)
         .invalidateHttpSession(true)
         .addLogoutHandler(logoutHandler)
         .deleteCookies(cookieNamesToClear)
         .and()  ...
}
```

Spring Security Taglib

- Spring Security ofrece su propio taglib que permite a los archivos JSP acceder a información de seguridad y personalizar la presentación:

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

- Es necesario agregar la dependencia:

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

- Se compone de tres tags o etiquetas:
  - authorize: Evalua una expresión de control de acceso..
  - authentication: Da acceso al objeto Authentication que contiene el objeto UserDetails.
  - accesscontrollist: Utilizado con el módulo Access Control List (ACL)

```
<sec:authorize access="isAnonymous()">
    <a class="nav-link" href="${pageContext.request.contextPath}/mylogin">Log In</a>
</sec:authorize>
<sec:authorize access="isAuthenticated()">
    <sec:authentication property="principal.username" />
    <a class="nav-link" href="${pageContext.request.contextPath}/logout">Log Out</a>
</sec:authorize>
```


JWT: JSON Web Tokens

https://jwt.io

- JSON Web Token (JWT) es un estándar abierto (RFC-7519) basado en JSON para crear un token que sirva para enviar datos entre aplicaciones o servicios y garantizar que sean válidos y seguros.
- El caso más común de uso de los JWT es para manejar la autenticación en aplicaciones móviles o web. Para esto cuando el usuario se quiere autenticar manda sus datos de inicio del sesión al servidor, este genera el JWT y se lo manda a la aplicación cliente, posteriormente en cada petición el cliente envía este token que el servidor usa para verificar que el usuario este correctamente autenticado y saber quien es.
- Se puede usar con plataformas IDaaS (Identity-as-a-Service) como Auth0 que eliminan la complejidad de la autenticación y su gestión.
- También es posible usarlo para transferir cualquier datos entre servicios de nuestra aplicación y asegurarnos de que sean siempre válido. Por ejemplo si tenemos un servicio de envío de email otro servicio podría enviar una petición con un JWT junto al contenido del mail o cualquier otro dato necesario y que estemos seguros que esos datos no fueron alterados de ninguna forma.

JavaScript Object Notation
http://tools.ietf.org/html/rfc4627

AENDICE: JSON

Introducción

- JSON (JavaScript Object Notation) es un formato sencillo para el intercambio de información.
- El formato JSON permite representar estructuras de datos (arrays) y objetos (arrays asociativos) en forma de texto.
- La notación de objetos mediante JSON es una de las características principales de JavaScript y es un mecanismo definido en los fundamentos básicos del lenguaje.
- En los últimos años, JSON se ha convertido en una alternativa al formato XML, ya que es más fácil de leer y escribir, además de ser mucho más conciso.
- No obstante, XML es superior técnicamente porque es un lenguaje de marcado, mientras que JSON es simplemente un formato para intercambiar datos.
- La especificación completa del JSON es la RFC 4627, su tipo MIME oficial es application/json y la extensión recomendada es .json.

Estructuras

- JSON está constituido por dos estructuras:
  - Una colección de pares de nombre/valor. En los lenguajes esto es conocido como un objeto, registro, estructura, diccionario, tabla hash, lista de claves o un arreglo asociativo.
  - Una lista ordenada de valores. En la mayoría de los lenguajes, esto se implementa como tablas, arreglos, vectores, listas o secuencias.
- Estas son estructuras universales; virtualmente todos los lenguajes de programación las soportan de una forma u otra. Es razonable que un formato de intercambio de datos que es independiente del lenguaje de programación se base en estas estructuras.

Sintaxis

- Un array es un conjunto de valores separados por comas (,) que se encierran entre corchetes […]
- Un objeto es un conjunto de pares nombre:valor separados por comas (,) que se acotan entre llaves { … }
- Los nombre son cadenas, entre comillas dobles (").
- El separador entre el nombre y el valor son los dos puntos (:)
- El valor debe ser un objeto, un array, un número, una cadena o uno de los tres nombres literales siguientes (en minúsculas):
  - true, false o null
- Se codifica en Unicode, la codificación predeterminada es UTF-8.

Valores numéricos

- La representación de números es similar a la utilizada en la mayoría de los lenguajes de programación.
- Un número contiene una parte entera que puede ser prefijada con un signo menos opcional, que puede ser seguida por una parte fraccionaria y / o una parte exponencial.
- La parte fraccionaria comienza con un punto (como separador decimal) seguido de uno o más dígitos.
- La parte exponencial comienza con la letra E en mayúsculas o minúsculas, lo que puede ser seguido por un signo más o menos, y son seguidas por uno o más dígitos.
- Los formatos octales y hexadecimales no están permitidos. Los ceros iniciales no están permitidos.
- No se permiten valores numéricos que no se puedan representar como secuencias de dígitos (como infinito y NaN).

Valores cadena

- La representación de las cadenas es similar a las convenciones utilizadas en la familia C de lenguajes de programación.
- Una cadena comienza y termina con comillas (").
- Se pueden utilizar todos los caracteres Unicode dentro de las comillas con excepción de los caracteres que se deben escapar: los caracteres de control (U + 0000 a U + 001F) y los caracteres con significado.
- Cuando un carácter se encuentra fuera del plano multilingüe básico (U + 0000 a U + FFFF), puede ser representado por su correspondiente valor hexadecimal. Las letras hexadecimales A-F puede ir en mayúsculas o en minúsculas.
- Secuencias de escape:
  - \\, \/, \", \n, \r, \b, \f, \t
  - \u[0-9A-Fa-f]{4}

Objeto con anidamientos

```
{
      "Image": {
          "Width":  800,
          "Height": 600,
          "Title":  "View from 15th Floor",
          "Thumbnail": {
              "Url":    "/image/481989943",
              "Height": 125,
              "Width":  "100"
              },
          "IDs": [116, 943, 234, 38793]
          }
  }
```


Array de objetos

```
[
    {
       "precision": "zip",
       "Latitude":  37.7668,
       "Longitude": -122.3959,
       "City":      "SAN FRANCISCO",
       "State":     "CA",
       "Zip":       "94107"
    },
    {
       "precision": "zip",
       "Latitude":  37.371991,
       "Longitude": -122.026020,
       "City":      "SUNNYVALE",
       "State":     "CA",
       "Zip":       "94085"
    }
  ]
```

JSON en JavaScript

- El Standard Built-in ECMAScript Objects define que todo interprete de JavaScript debe contar con un objeto JSON como miembro del objeto Global.
- El objeto debe contener, al menos, los siguientes miembros:
  - **JSON.parse** (Función): Convierte una cadena de la notación de objetos de JavaScript (JSON) en un objeto de JavaScript.
  - **JSON.stringify** (Función): Convierte un valor de JavaScript en una cadena de la notación de objetos JavaScript (JSON).