



Desarrollo en Entorno Servidor

Apuntes

Índice

TEMA 1: INTRODUCCIÓN	4
FUNCIONAMIENTO DE UNA APLICACIÓN WEB	4
PROTOCOLO HTTP	5
BACKEND VS. FRONTEND	7
ARQUITECTURA DE APLICACIONES	9
SPRING FRAMEWORK	13
SPRINGBOOT	15
HERRAMIENTAS PARA DESARROLLADORES	17
VIDEOS DE APOYO	18
TEMA 2: INSTALACIÓN Y PRIMER PROYECTO.....	19
INSTALAR JDK	19
VISUAL STUDIO CODE	19
PRIMER PROYECTO	23
CONTENIDO ESTÁTICO	29
APACHE NETBEANS	30
TEMA 3: CONTROLADORES Y VISTAS	33
CARACTERÍSTICAS DEL CONTROLADOR	33
THYMELEAF Y CONTENIDO DINÁMICO	35
PASO DE DATOS A LA PLANTILLA	40
RESCATAR PARÁMETROS DE LAS URL	41
RETORNO DE LOS MÉTODOS DEL CONTROLADOR.....	44
CONSTRUCCIÓN DE URLs DINÁMICAS EN LA VISTA.....	45
RESUMEN DE CONTROLADOR + VISTA	47
TEMA 4: SERVICIOS.....	50
DEFINICIÓN DE SERVICIO	50
GESTIÓN DE ERRORES.....	52
SERVICIOS MEDIANTE INTERFACES Y CLASES	55
COMMANDLINERUNNER	56
TEMA 5: FORMULARIOS	58
DESARROLLO DE FORMULARIOS	58
FORMULARIOS DE EDICIÓN.....	62
VALIDACIÓN DE FORMULARIOS	62
SUBIDA DE FICHEROS	64
ALMACENAMIENTO DE FICHEROS.....	65
ENVÍO DE EMAILS	66
TEMA 6: MODELO Y REPOSITORIOS	70
ENTIDADES	70
LOMBOK	70
REPOSITORIOS.....	72
CRUD CON REPOSITORIOS EN MEMORIA	73
LECTURA DE PARÁMETROS DE FICHEROS	81
TEMA 7: ACCESO A DATOS.....	83
INTRODUCCIÓN.....	83
CONFIGURACIÓN INICIAL DE H2	85
ENTIDADES	86
REPOSITORIOS	88
INTERFACES REPOSITORY	88
MAPEO DE ASOCIACIONES	96
DTO	115
OTROS CONCEPTOS JPA IMPORTANTES	117

TEMA 8: SEGURIDAD Y CONTROL DE ACCESO	126
INTRODUCCIÓN.....	126
CONFIGURACIÓN BÁSICA	127
GESTIÓN DE USUARIOS.....	130
RESUMEN PARA SECURIZAR NUESTRA APLICACIÓN.....	136
OTRAS OPCIONES SOBRE AUTENTIFICACIÓN	136
TEMA 9: API REST	139
INTRODUCCIÓN.....	139
PROYECTO API REST	142
GESTIÓN DE ERRORES.....	150
ELEMENTOS AVANZADOS.....	155
OPENAPI Y SWAGGER	163
CONSUMIENDO APIs.....	167
SEGURIDAD EN API REST.....	174
TEMA 10: PASE A PRODUCCIÓN Y TESTING.....	185
INTRODUCCIÓN.....	185
EMPAQUETADO “WAR”.....	185
MySQL	187
TESTING	193
PERFILES DE CONFIGURACIÓN	202
LOGGING.....	203
MONITORIZACIÓN CON ACTUATOR.....	207
DESPLIEGUE (DEPLOYING)	209
TAREAS PLANIFICADAS.....	217
MICROSERVICIOS	218
ENLACES DE INTERÉS	223

Fernando Rodríguez Diéguez
rdf@fernandowirtz.com
Versión 2024-07-12

Tema 1: Introducción

En este primer tema presentaremos conceptos generales a modo de introducción sobre programación web, programación del lado del servidor o *Back-End* y el framework Spring con el que trabajaremos a lo largo de todo el curso. Los conceptos de la parte final de este primer tema sobre el funcionamiento de Spring son bastante abstractos y hasta que no los pongamos en práctica puede que no los entendamos completamente. Será aconsejable releer este tema de nuevo más adelante, después de haber realizado nuestras primeras aplicaciones con Spring.

En el segundo tema realizaremos la instalación y configuración del entorno de trabajo (IDE, JDK, etc.) y en los temas del 3 al 6 aprenderemos a desarrollar aplicaciones completas, esto es, con parte de servidor, pero también con una interfaz de usuario a la que se podrá acceder a través del navegador, por lo que serán totalmente operativas. Para esta parte emplearemos Spring MVC y Thymeleaf como motor de plantillas.

En el tema 7 veremos cómo trabajar con distintas bases de datos, de una forma transparente para nosotros gracias a Spring Data.

El tema 8 es muy importante, ya que “restificaremos” nuestras aplicaciones, es decir, pasaremos de aplicaciones completas (con parte cliente y servidor) a aplicaciones de servidor “puras” que responderán a tipos de clientes totalmente diferentes mediante API Rest.

En el tema 9 veremos cómo gestionar la seguridad en nuestras aplicaciones: gestión de usuarios y permisos, sesiones, etc. y dejaremos para el tema 10 otras áreas importantes como el pase a producción y el testing, etc.

Funcionamiento de una aplicación web

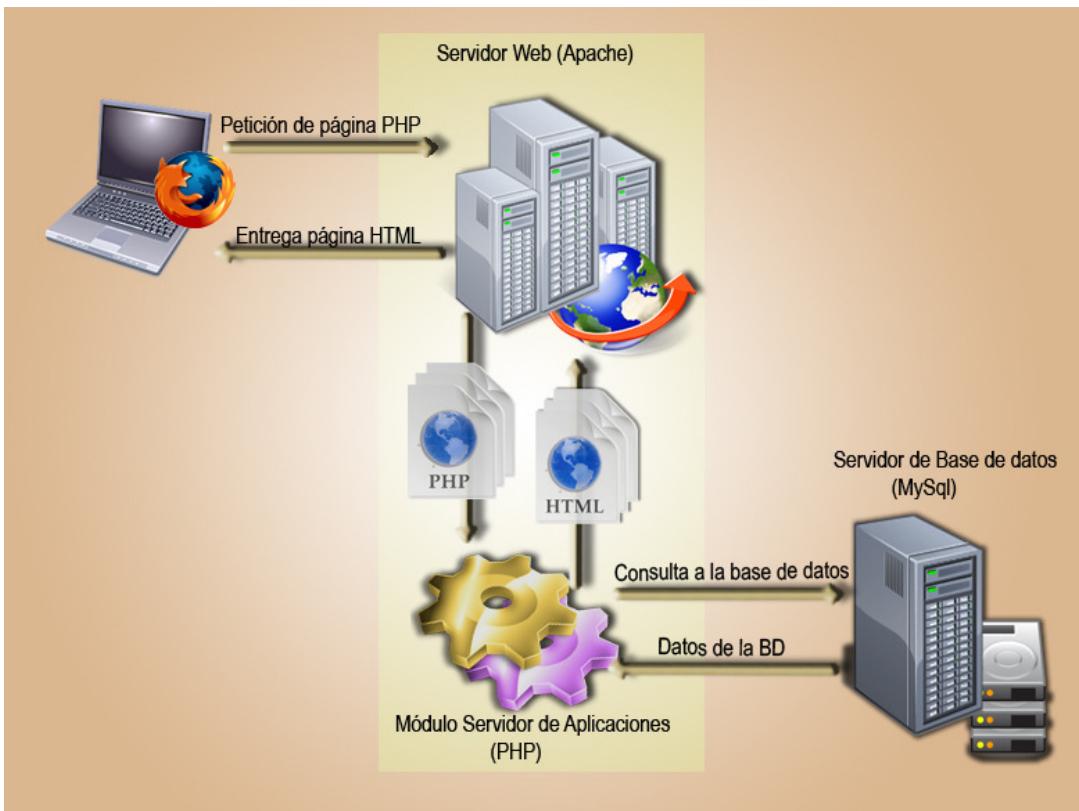
Las aplicaciones web funcionan siguiendo el denominado modelo cliente-servidor, siendo el cliente un programa o dispositivo que inicia una comunicación (una petición) y el servidor el que responde a dicha petición con los datos solicitados.

Uno de los servidores web más populares es **Apache** HTTP Server Project. En el caso del cliente, frecuentemente será un **navegador web**, pero pueden ser otros como apps de dispositivos móviles, dispositivos *Internet of Things* (IoT), etc. El protocolo de comunicación más empleado para la comunicación entre el cliente y el servidor es HTTP. A través de internet y mediante este protocolo el servidor será accesible y será identificado por un dominio, que generalmente empieza por www. El ordenador u ordenadores que actúan como servidor estarán “a la escucha” de las peticiones que lleguen por un determinado puerto (tradicionalmente el puerto 80).

En el caso de una aplicación web estática clásica, el servidor contendrá una carpeta donde tendrá todos los archivos HTML, y los clientes harán peticiones de esas páginas mediante una URL (la URL contiene el protocolo, el dominio y la página/carpeta solicitada). Si el servidor encuentra esa página se la devolverá al usuario, que en su navegador web *renderizará* para mostrar su contenido. Esas peticiones serán de archivos HTML, CSS, JavaScript y otros recursos como pueden ser archivos multimedia.



Este es un esquema muy simple ya que no permite contenido dinámico, siempre se mostraría el mismo contenido. Si deseamos añadir contenido dinámico (por ejemplo, tratar los datos enviados por el usuario en un formulario, realizar algún cálculo en el servidor, etc) necesitamos incorporar a ese esquema el denominado "servidor de aplicaciones" que mediante algún lenguaje de programación en el servidor (Java, PHP, C#...) se encargaría de construir la página HTML de forma dinámica. Si en ese tratamiento, queremos acceder a una base de datos para consultar o modificar datos necesitamos un gestor de base de datos (MySQL/MariaDB, Oracle, SQLServer, MongoDB, etc.)



En la imagen anterior se muestra el esquema que acabamos de describir, mediante lo que se conoce como una pila AMP (por las siglas: de Apache, MySQL, PHP). Existen varios paquetes comerciales con esta pila, siendo **XAMPP** uno de los más populares, gratuito y multiplataforma.

Este esquema tradicional de construir la página en el servidor de forma mixta, mezclando etiquetas HTML con los datos dinámicos para formar la página completa, también está cayendo en desuso últimamente, ya que en la actualidad se distinguen dos aplicaciones diferenciadas: la del cliente y la del servidor. La aplicación en el servidor no construye él mismo la página HTML, sino que simplemente obtiene los datos necesarios (API) y se los pasa a la aplicación cliente en formato JSON, XML, etc. sin maquetado. La aplicación cliente (que puede ser HTML pero también una app de dispositivo móvil, o cualquier otra tecnología) es la que se encargará de construir la vista final que mostrará el navegador. En apartados posteriores hablaremos más de esto diferenciando aplicación *Front-End y Back-End*.

Protocolo HTTP

El Protocolo de transferencia de hipertexto (en inglés, *Hypertext Transfer Protocol*, abreviado HTTP) es un protocolo de comunicación que permite las transferencias de información a través de archivos (HTML, XML, JSON...) entre dispositivos a través de internet.

Es un protocolo orientado a transacciones y sigue el esquema petición-respuesta entre el cliente y el servidor. El cliente (se le suele llamar "agente de usuario", del inglés *user agent*) realiza una petición enviando un mensaje con cierto formato al servidor y este responderá con otro mensaje.

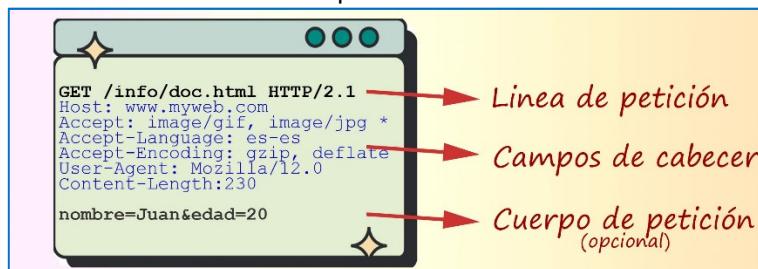
HTTP es un protocolo sin estado, por lo que no guarda ninguna información sobre conexiones anteriores por lo que si necesitamos mantener el estado necesitaremos implementarlo con alguna técnica específica como pueden ser las *cookies*.

HTTPS (HTTP seguro) es la versión segura del http, una variante del mismo protocolo que se basa en la creación de un canal cifrado para la transmisión de la información, lo cual lo hace más apropiado intercambio de datos más sensibles (como claves y usuarios personales).

Estructura de los Mensajes

De manera general, una petición HTTP se divide en 3 partes:

- **Línea de petición:** incluye el método HTTP a utilizar, URL del recurso solicitado y versión de protocolo. El método (o comando o mensaje o verbo) indica el tipo de acción (los veremos en detalle en el siguiente apartado). Ejemplo: `GET /doc/index.html HTTP/2.1`
- **Campos de cabecera:** son metadatos sobre la petición, como el nombre del servidor, gestión de la cache, navegador empleado, idioma, tipo de contenidos, etc. Cada va en una línea, especificando el tipo de metadato, dos puntos, un espacio en blanco y el valor asignado.
- **Cuerpo (opcional):** representa el contenido de lo que queremos enviar, en el caso de un formulario, incluiría los datos introducidos por el usuario.



La respuesta HTTP tiene una estructura similar:

- Línea de status: Indica el estado resultante de la petición.
- Cabecera: similar al de la petición, metadatos sobre la respuesta.
- Cuerpo: es la parte más importante ya que incluirá los datos solicitados por el cliente, en formato HTML, JSON, etc.

Ejemplo:

```
HTTP/1.1 200 OK
Date: Thu, 03 Jan 2022 23:26:07 GMT
Server: gws
Accept-Ranges: bytes
Content-Length: 68894
Content-Type: text/html; charset=UTF-8

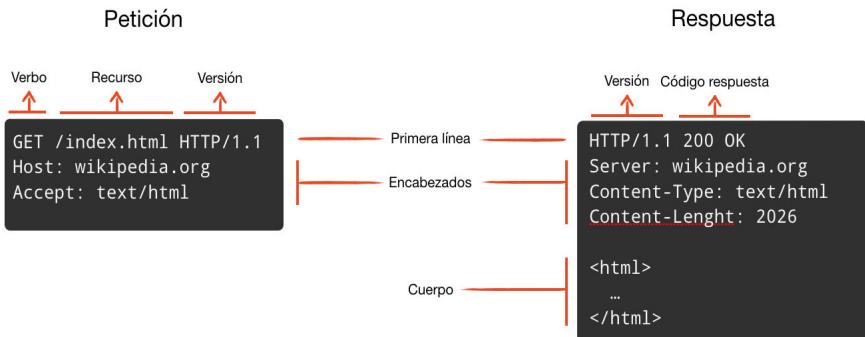
<!doctype html><html . . .
```

En la primera línea (status), la respuesta fue un 200 OK. Luego tenemos las cabeceras de la respuesta. Finalmente, separado por una línea en blanco de las cabeceras, tenemos el cuerpo de la respuesta, que en este caso es un documento HTML.

Estados de respuesta más usuales:

- **1xx:** Mensaje informativo.
- **2xx:** Exito
 - 200 OK
 - 201 Created
 - 202 Accepted
 - 204 No Content
- **3xx:** Redirección
 - 300 Multiple Choice
 - 301 Moved Permanently
 - 302 Found
 - 304 Not Modified
- **4xx:** Error del cliente
 - 400 Bad Request
 - 401 Unauthorized
 - 403 Forbidden
 - 404 Not Found
- **5xx:** Error del servidor
 - 500 Internal Server Error
 - 501 Not Implemented
 - 502 Bad Gateway
 - 503 Service Unavailable

Este sería el esquema completo:



Métodos de petición

HTTP define una serie predefinida de métodos de petición (algunas veces referido como "verbos") que pueden utilizarse. Cada método indica la acción que desea que se efectúe sobre el recurso identificado por la URI, por ejemplo, el recurso puede corresponderse con un archivo que reside en el servidor. El número de métodos de petición ha aumentado al avanzar las versiones. Los más habituales son:

GET: El método GET solicita una representación del recurso especificado. Las solicitudes que usan GET solo deben recuperar datos y no deben tener ningún otro efecto.

HEAD: Pide una respuesta idéntica a la que correspondería a una petición GET, pero en la respuesta no se devuelve el cuerpo. Esto es útil para poder recuperar los metadatos de los encabezados de respuesta, sin tener que transportar todo el contenido.

POST: Envía datos para que sean procesados por el recurso identificado en la URI, como por ejemplo al enviar un formulario. Los datos se incluirán en el cuerpo de la petición. A nivel semántico está orientado a crear un nuevo recurso, cuya naturaleza vendrá especificada por la cabecera *Content-Type*.

- Para datos formularios codificados como una URL (aunque viajan en el cuerpo de la petición, no en la URL): *application/x-www-form-urlencoded*
- Para bloques a subir, ej. ficheros: *multipart/form-data*
- Además de los anteriores, no hay un estándar obligatorio y también podría ser otros como *text/plain*, *application/json*, *application/octet-stream*...

PUT: Envía datos al servidor, pero a diferencia del método POST la URI de la línea de petición no hace referencia al recurso que los procesará, sino que identifica a los propios datos. Otra diferencia con POST es semántica, mientras que POST está orientado a la creación de nuevos contenidos, PUT está más orientado a la actualización de los mismos (aunque también podría crearlos).

DELETE: Borra el recurso especificado.

Existen otros comandos como TRACE, OPTIONS, CONNECT, etc.

Backend vs. FrontEnd

Front-End

Como ya adelantamos previamente, *Front-End* es la parte de la aplicación que interactúa con los usuarios, y también es conocida como "*el lado del cliente*". Básicamente es todo lo que vemos en la pantalla cuando accedemos a un sitio web o aplicación: tipos de letra, colores, disposición para distintos tipos de dispositivos (diseño *responsive*), interacción con el usuario, efectos visuales, etc. Este conjunto crea la experiencia del usuario.

Los lenguajes de programación de este entorno, siempre hablando de entorno web, son: HTML5, CSS3 y JavaScript (y su evolución TypeScript). Es también frecuente el uso de librerías como Ajax (también JQuery, aunque cayendo en desuso actualmente) para comunicación asíncrona con el servidor, de

forma que podamos obtener ciertos datos para actualizar una página sin tener que solicitarla de nuevo de forma completa al servidor.



Cabe señalar que hay multitud de frameworks basados en JavaScript/TypeScript que facilitan la labor del desarrollo de estas aplicaciones Front-End; entre las más populares citamos: Angular (de Google), React (de Facebook) o Vue.



Back-End

Back-End se refiere al interior de las aplicaciones que viven en el servidor y al que a menudo se le denomina "el lado del servidor". En el servidor se reciben las peticiones desde el cliente, se procesan realizando las operaciones necesarias (matemáticas, lógicas, etc) y accediendo a los distintos repositorios de datos (fichero, bases de datos, etc...) y se devuelve una respuesta al aplicativo del *Front-End*.

Los lenguajes más habituales en el *Back-End* son Java, PHP, Python, C#, Ruby, etc. También se ha incorporado a esta categoría JavaScript, lenguaje típico de *Front-End* pero que bajo un servidor *Node.js* es capaz de ejecutarse como lenguaje de servidor.

En cuanto a los frameworks: **Spring** es el más utilizado en Java, **Laravel** y **Simphony** sobre PHP, **Django** y **Flask** sobre Python, **ASP.NET** con C# en entorno Microsoft, etc.



Servidor de Base de Datos

Otro punto importante en el servidor son los gestores de bases de datos que mantendrán la persistencia de los datos de las aplicaciones. Existen actualmente dos paradigmas: el relacional clásico con gestores como *SQLServer*, *MySQL/MariaDB*, *Oracle*, etc. y el modelo NoSQL (*not only SQL*) siendo *MongoDB* su máximo exponente.



Aplicaciones móviles

Para cerrar el conjunto de plataformas dentro del desarrollo de software actual, tenemos que dedicarles un apartado a las aplicaciones para dispositivos móviles, dado su auge entre los usuarios. En este ámbito tenemos dos mundos totalmente diferenciados: el sistema operativo Android y el sistema operativo IOS; para el primero el lenguaje nativo más empleado es Kotlin (y también Java) y para el segundo es Swift (y también Objetive C). En cuanto a frameworks en el mercado, su objetivo principal misión es la reutilización de código, de forma que con un solo desarrollo se puedan crear aplicaciones que se puedan ejecutar en ambos sistemas. Han surgido muchas a lo largo de estos años: Cordova, Xamarin, Ionic, etc... siendo las dos más populares en este momento Flutter y React Native.



Full-Stack

Es frecuente en las ofertas de trabajo para desarrolladores de software distinguir los dos ámbitos: *Front-End y Back-End*, mientras que para referirse ambos se emplea el término *Full-Stack*.

Arquitectura de Aplicaciones

En este apartado vamos a ver distintos enfoques a la hora de organizar las aplicaciones de servidor, teniendo en cuenta sus ventajas e inconvenientes, especialmente en los que se refiere al escalado de la aplicación (nuevas funcionalidades y asignación de más recursos), facilidad de mantenimiento, testing, etc.

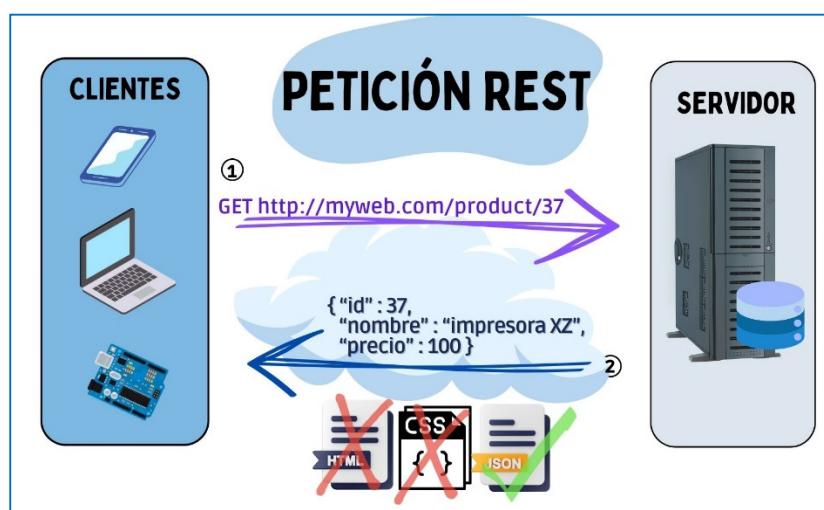
Aplicaciones Web vs API REST

El sistema clásico de desarrollo de aplicaciones web, todo el procesamiento se sitúa en el servidor: al código HTML (plantilla) se le añaden *"variables"* que, cuando se compone la página, son sustituidas por valores dinámicos, como por ejemplo datos extraídos de la base de datos. Al navegador cliente le llega la página web definitiva (una vista) y solo tiene que mostrarla. Este esquema lo utilizaban las antiguas páginas JSP y actualmente Spring con su módulo Spring MVC y con un motor de plantillas como Thymeleaf. *Trabajaremos del capítulo 3 al 8 con este esquema.*

Un planteamiento totalmente diferente es API REST en el servidor se despreocupa de la parte cliente, en el cliente puede haber aplicación móvil, un navegador, IoT, etc.; servidor lo que hace es enviar los datos necesarios y estos son tratados por el cliente como él decida. El protocolo utilizado para esta comunicación suele ser HTTP y el intercambio de datos se suele hacer generalmente mediante archivos JSON. *Trabajaremos con este esquema en el tema 9.*

Ventajas e inconvenientes de este sistema respecto al tradicional:

- El renderizado en el servidor es un esquema más sencillo ya que solo tenemos una aplicación, la del servidor, mientras que con REST tenemos una en servidor y otra en cliente, y por tanto más tecnologías involucradas.
- El renderizado en servidor es más rápido que el segundo ya el cliente no necesita ningún procesamiento, solo mostrar la página.
- El renderizado en servidor es mucho más rígido: solo responde páginas web, pero no peticiones de otros dispositivos: móviles, IoT, etc. Con REST el servidor ofrece solo los datos que son respuesta a las peticiones y es compatible con cualquier dispositivo que use HTTP.
- Con REST las aplicaciones tienen menos acoplamiento, están mejor estructuradas y son mucho más fáciles de testear y ampliar.
- Para el aprendizaje es más fácil de ver los resultados con el primer paradigma. Este motivo es el que nos llevará a usar Spring MVC y Thymeleaf en los primeros capítulos para ver en API REST en los últimos.



Aplicaciones monolíticas vs. Microservicios

Antiguamente las aplicaciones eran lo que ahora denominamos '**monolíticas**': esto es, aplicaciones completas con una fuerte cohesión y dependencia entre todos sus componentes en términos de su comportamiento. Toda ella se implementa como una única unidad y si es necesario escalar horizontalmente este tipo de aplicación, se debe duplicar la aplicación completa en varios servidores.

Cuando aumenta la complejidad de las aplicaciones, una forma de optimizarlas consiste en dividirlas en capas, de forma que cada capa tenga una responsabilidad diferente. Este enfoque sigue el principio de separación de intereses y ayuda a mantener organizado el código, de forma que los desarrolladores puedan encontrar fácilmente dónde se implementa una función determinada. Pero la arquitectura en capas ofrece una serie de ventajas que van más allá de la simple organización del código.

Al organizar el código en capas, la funcionalidad común de bajo nivel se puede reutilizar en toda la aplicación. Además, las aplicaciones pueden aplicar restricciones sobre qué capas se pueden comunicar con otras capas. Esta arquitectura permite lograr la encapsulación: cuando se cambia o reemplaza una capa, el impacto en otras debería ser mínimo, eso es logra por ejemplo con programación basada en interfaces e inyección de dependencias. [API Rest elimina la capa de presentación de la aplicación del servidor](#).

Las capas (y la encapsulación) facilitan considerablemente el reemplazo de funcionalidad dentro de la aplicación. Por ejemplo, es posible que una aplicación use inicialmente una base de datos ligera (SQLite, H2), pero más adelante cambie a una base de datos más potente como Oracle, o una persistencia basada en la nube, etc. Si la aplicación ha encapsulado correctamente su implementación de persistencia dentro de una capa con las interfaces públicas adecuadas, esa capa específica de persistencia sería fácilmente reemplazada con otras implementaciones de esas interfaces.

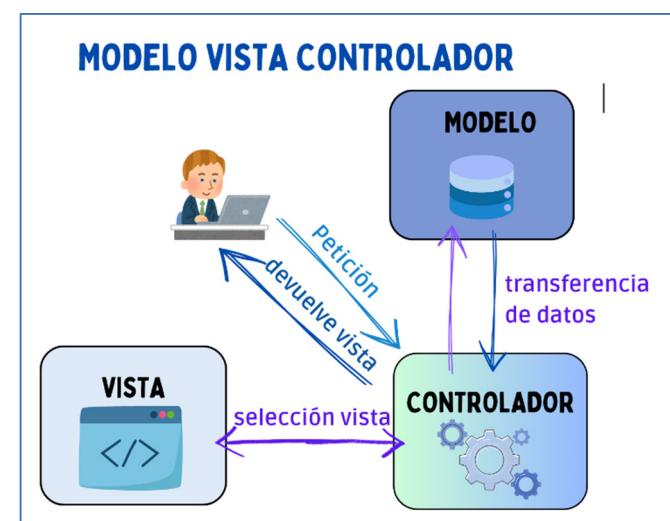
Por último, el testing es mucho más sencillo en una aplicación organizada en capas, ya que podemos realizar pruebas unitarias de una funcionalidad de una capa simulando fácilmente las entradas o salidas de las capas con las que interactúa. La prueba unitaria de una funcionalidad en una aplicación sin capas implica que la prueba incluya desde la interfaz de usuario hasta el acceso a la base de datos.

Un primer enfoque de esta división en capas (aunque bastante pobre) es el patrón MVC del que hablamos a continuación. Una aplicación dividida en capas, aunque presenta las mejoras comentadas, sigue siendo monolítica, plasmada en un solo proyecto, una sola unidad a compilar, distribuir y escalar. El siguiente nivel de optimización de diseño son los microservicios que describiremos más adelante.

MVC

Modelo-vista-controlador (MVC) es un patrón desarrollo que separa los datos y la lógica de negocio de la aplicación de su representación y el módulo encargado de gestionar los eventos. De manera genérica, los componentes de MVC se podrían definir como sigue:

Modelo: Es la representación de la información con la cual el sistema opera, por lo tanto, gestiona todos los accesos a dicha información, tantas consultas como actualizaciones (lógica de negocio). Envía a la 'vista' aquella parte de la información que en cada momento se le solicita para que sea mostrada (típicamente a un usuario). Las peticiones de acceso o manipulación de información llegan al 'modelo' a través del 'controlador'.



Controlador: Responde a eventos (usualmente acciones del usuario) e invoca peticiones al 'modelo' cuando se hace alguna solicitud sobre la información (por ejemplo, realizar un cálculo o modificar un registro en una base de datos). También puede enviar comandos a su 'vista' asociada si se solicita un cambio en la forma en que se presenta el 'modelo' (por ejemplo, desplazamiento o scroll por un documento o por los diferentes registros de una base de datos), por tanto, se podría decir que el 'controlador' hace de intermediario entre la 'vista' y el 'modelo'.

Vista: Presenta el 'modelo' (información y lógica de negocio) en un formato adecuado para interactuar (usualmente la interfaz de usuario), por tanto, requiere de dicho 'modelo' la información que debe representar como salida.

El proceso sería algo así:

1. El usuario interactúa con la interfaz de usuario de alguna forma (por ejemplo, pulsando un botón).
2. El controlador recibe la notificación de la acción solicitada por el usuario y accede al modelo, actualizándolo, posiblemente modificándolo de forma adecuada a la acción solicitada por el usuario (por ejemplo, el controlador actualiza el carro de la compra del usuario).
3. El controlador delega a los objetos de la vista la tarea de desplegar la interfaz de usuario. La vista obtiene sus datos del modelo para generar la interfaz apropiada para el usuario donde se reflejan los cambios en el modelo (por ejemplo, produce un listado del contenido del carro de la compra). El modelo no debe tener conocimiento directo sobre la vista.
4. La interfaz de usuario espera nuevas interacciones del usuario, comenzando el ciclo nuevamente.

Este sistema es bueno como primera aproximación, pero se queda "corto" en cuanto a la división de responsabilidades, por ejemplo, no queda clara una división entre las entidades que forman el sistema y las reglas de negocio.

Microservicios

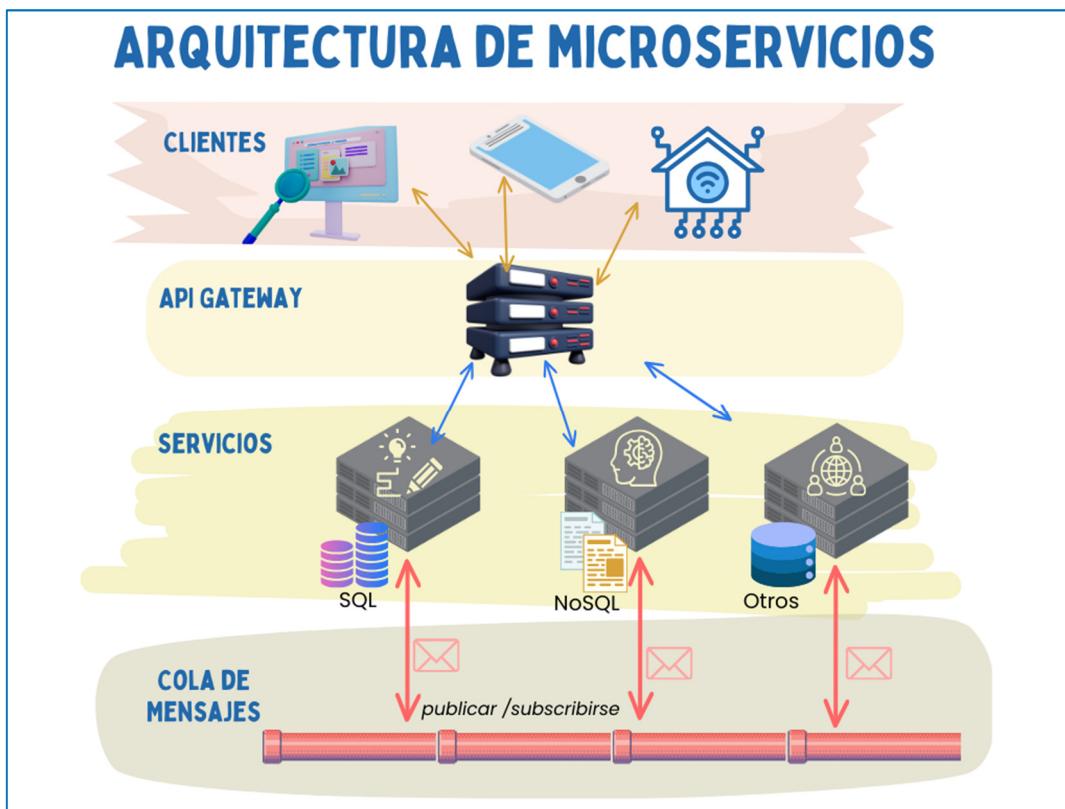
Los microservicios son el enfoque opuesto a las aplicaciones monolíticas ya que se basan en que una aplicación es un conjunto de servicios independientes. La gran diferencia es que cada microservicio es un artefacto software independiente y autónomo (se compila e instala por separado, puede ejecutarse en un servidor distinto al de otros servicios de su misma aplicación global, se puede escalar sin afectar el funcionamiento de otros servicios etc.) Un microservicio debe ser especializado (enfocado en resolver un problema específico), por ejemplo: gestionar el login de usuario, consultar un producto, etc.

Esa independencia entre servicios hace que estas aplicaciones sean más fáciles de escalar (un servicio crítico puede estar balanceado en más servidores que otro menos usado) y más rápidas de desarrollar (no se ven afectados otros servicios).

El inconveniente principal de esta arquitectura es mantener una organización clara cuando el número de microservicios de la aplicación es elevado. La comunicación desde los clientes es a través de un elemento común, al que denominamos API Gateway que conoce todos los servicios desplegados y cómo llegar a ellos. Será el encargado de redirigir las peticiones al microservicio adecuado.

Los microservicios se comunican entre sí mediante una cola de mensajes donde publican sus peticiones a otros servicios y reciben las peticiones de otros servicios (RabbitMQ, Apache Kafka).

En las implementaciones reales pueden intervenir más actores para el correcto funcionamiento del sistema, pero en el siguiente gráfico se presenta una estructura básica de esta arquitectura.



Arquitecturas limpias

Una evolución del patrón MVC son las divisiones en capas que se emplean actualmente y que frecuentemente se denominan “arquitecturas limpias”. Dentro de ellas, es común hablar de Arquitectura Hexagonal y Arquitectura DDD. Estos principios se pueden aplicar tanto a aplicaciones monolíticas como basadas en microservicios.

Estas arquitecturas se basan en una serie de principios que debe cumplir nuestro código:

- Ser **independiente del framework** (no deben influir en nuestro modelo).
- Ser **testable**. Debemos poder probar nuestras reglas de negocio sin pensar en base de datos, interface gráfica u otros componentes no esenciales de nuestro sistema.
- Ser **independiente de la UI**. Si la UI cambia a menudo esto no puede afectar al resto de nuestro sistema, que tiene que ser independiente.
- Ser **independiente de la base de datos**. Deberíamos poder cambiar de Oracle, a SQLServer, a MongoDB, a Casandra o a cualquier otra base de datos sin que afectara demasiado a nuestro sistema.
- Ser **independiente de cualquier entidad externa**. No deberíamos saber nada de entidades externas, por lo que no deberemos depender de ellas.



Entidades: Las entidades son las que incluyen las reglas de negocio críticas para el sistema. Estas entidades pueden ser utilizadas por distintos componentes de la arquitectura, por lo que son independientes, y no deben cambiar a consecuencia de otros elementos externos. Una entidad deberá englobar un concepto crítico para el negocio, y nosotros tendremos que separarlo lo

más posible del resto de conceptos. Esa entidad recibirá los datos necesarios, y realizará operaciones sobre ellos para conseguir el objetivo deseado.

Casos de uso: En este caso nos encontramos con las **reglas de negocio aplicables a una aplicación concreta**. Estos casos de uso siguen un flujo para conseguir que las reglas definidas por las entidades se cumplan. Los casos de uso, solo definen cómo se comporta nuestro sistema, definiendo los datos de entrada necesarios, y cuál será su salida. Los cambios en esta capa no deberían afectar a las entidades, al igual que los cambios en otras capas externas no deberían afectar a los casos de uso. Es importante que no pensemos en como los datos que genera un caso de uso serán presentados al usuario. No deberemos pensar en HTML, o en SQL. Un caso de uso recibe datos estructurados y devuelve más datos estructurados.

Adaptadores de interfaz: Los datos generados por los casos de uso y las entidades, tienen que transformarse en algo entendible por la siguiente capa que los va a utilizar y de eso se encarga esta capa. Pensando en MVC, por ejemplo, los controladores y las vistas, pertenecerían a esta capa, y el modelo, serían los datos que se pasan entre los casos de uso y los controladores para luego poder presentar las vistas. Lo mismo aplicaría para, por ejemplo, presentar información a un servicio externo, ya que en esta capa definiríamos la manera en la que el dato de las capas internas se presenta al exterior.

Frameworks y drivers: En la capa más externa es donde van los detalles. Y la base de datos es un detalle, nuestro framework web, es un detalle etc.

En definitiva, debemos pensar en nuestro sistema, como un sistema de plugins, de forma que los componentes estén aislados y podamos sustituir unos por otros sin demasiados problemas. Estos enfoques tienen ventajas fundamentales:

- Facilidad de comprensión del problema y por tanto de su desarrollo.
- Independencia durante el ciclo de desarrollo: Por ejemplo, podemos empezar a desarrollar todas nuestras reglas de negocio, sin tener en cuenta su persistencia, ya que esa parte se realiza a través de una interface. Primero podemos utilizar objetos en memoria, y según avancemos, ir añadiendo sistemas más sofisticados. Al final podremos elegir entre usar una base de datos relacional, NoSQL, o incluso guardar la información en archivos.
- Facilidad de testing.
- Facilitar escalabilidad y despliegue.
- Eliminar acoplamientos.

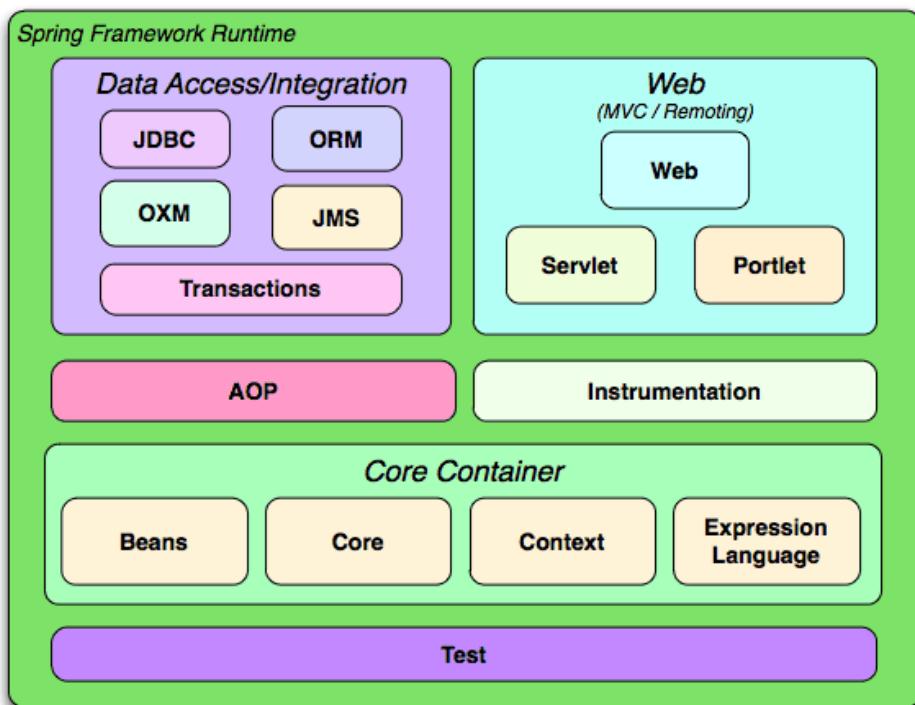
Si tienes interés en estos temas investiga lo que son los principios SOLID, por ejemplo en esta web: <https://devexperto.com/principios-solid/>

Spring Framework

Spring es un framework Open Source que facilita la creación de aplicaciones en Java, Kotlin y Groovy principalmente para el Back-End. Se compone de herramientas y utilidades que generan aplicaciones completas descargando al desarrollador de la gestión de aspectos internos de comportamiento y generando código automático para tareas estándar, siendo las más importantes: el acceso a base de datos, la gestión de la seguridad de la aplicación, generación de API Rest, etc.

Está dividido en diversos módulos que podemos utilizar, ofreciéndonos muchas más funcionalidades:

- **Core container:** es el núcleo del framework y proporciona entre otras cosas, inyección de dependencias e inversión de control.
- **Web:** nos permite crear controladores Web, tanto de vistas MVC como aplicaciones REST.
- **Acceso a datos:** abstracciones sobre JDBC, ORMs como Hibernate, sistemas OXM (*Object XML Mappers*), JSM y transacciones.
- **Programación orientada a Aspectos (AOP):** ofrece el soporte para aspectos.
- **Instrumentación:** proporciona soporte para la instrumentación de clases.
- **Pruebas de código:** contiene un *framework* de *testing*, con soporte para JUnit y TestNG y todo lo necesario para probar los mecanismos de Spring.



Uno de las características más importantes Spring es la **inyección de dependencias** y el **inversor de control**. Estos son unos conceptos un poco complejos para explicar teóricamente, pero los veremos en funcionamiento a lo largo del curso. Podemos decir que la inyección de dependencias es un patrón de software que se basa en que un elemento externo se encarga de la creación de los objetos a medida que son requeridos por otros objetos de nuestra aplicación.

Ese elemento externo que realiza esta tarea es el inversor de control (IoC). Como ejemplo, podemos decir que si en una aplicación Spring, una instancia de la clase *Alumno* necesita una instancia de la clase *Asignatura*, decimos que *Asignatura* es una dependencia de *Alumno* y será el IoC el encargado de crear esa asignatura en el alumno, en vez que crearla nosotros con su constructor mediante *new Asignatura()*. En Spring el contenedor IoC está representado por la interfaz *ApplicationContext*, la cual es la responsable de configurar e instanciar todos los objetos (los beans) y manejar su ciclo de vida.

Para terminar este apartado, hemos de decir que uno de los grandes inconvenientes de Spring es su profunda configuración, bien a través de ficheros XML, clases Java (Java Config) o anotaciones (estas últimas son las más empleadas en la actualidad). Para facilitar este trabajo tenemos el módulo de SpringBoot que realizará toda esa tediosa configuración por nosotros y que veremos a continuación.

SpringBoot

Como acabamos de comentar, la configuración con Spring es bastante compleja: tipo de proyecto (Maven vs. Gradle, Java vs. Kotkin vs. Groovy), configuración de Java, dependencias (¿Cuáles necesito? ¿versiones?, etc.), estructura de carpetas del proyecto, configuración y manejo de los componentes (beans), etc.

SpringBoot es un módulo dentro del ecosistema Spring Framework que va a facilitar toda esta configuración con un mínimo esfuerzo. Estas son sus características principales:

- Usa el patrón de diseño “Convención sobre configuración” (CoC) que se basa en minimizar las decisiones que tiene que tomar el desarrollador en cuanto a configuración, pero sin perder flexibilidad. Así el hace una configuración por defecto común a casi cualquier proyecto, pero nos permitirá cambiar algún parámetro para algún proyecto concreto. Por ejemplo, si tenemos una clase llamada Empleado, CoC, considerará que la tabla subyacente se llamará igual, aunque podremos cambiar este comportamiento.
- Permite crear aplicaciones Spring independientes. Tenemos la opción de que nuestro proyecto sea un archivo “jar” con un servidor embebido (Tomcat, Jetty...) o bien un archivo “war” que desplegaremos en un servidor web externo.
- Incluye dependencias “starter”. Estas dependencias incluyen a su vez otras dependencias transitivamente de forma que no tenemos que incluirlas una a una, y cada grupo de dependencias irán con una versión compatible. Por ejemplo, si añadimos la dependencia starter “web”, él incluirá todas las dependencias que necesite para hacer proyecto web.
- Configuración automática tanto de librerías propias como de terceros.
- No genera código ni configuración XML, es transparente para el desarrollador.
- Requisitos: JDK8 o superior, SpringBoot 2 trabaja sobre Spring 5 y SpringBoot 3 sobre Spring 6. También necesitamos Maven 3 o superior e incluye Tomcat 9 (ya embebido, no hay que instalarlo).
- Soporte para distintos IDE (Eclipse, Visual Studio Code, Netbeans, etc.)

Estereotipos

Spring dispone de un elemento llamado *ApplicationContext* que contiene las instancias de objetos que ha identificado para administrar su ciclo de vida completamente. Estos se llaman “beans”.

Como ya hemos comentado, utilizando el principio de inversión de control, Spring recopila instancias de los “beans” de nuestra aplicación y las gestiona en el momento adecuado. Podemos mostrar las dependencias de los beans a Spring sin necesidad de manejar la configuración y la creación de instancias de esos objetos.

La forma de indicarle a Spring cuales de nuestras clases son beans que él debe registrar en el *ApplicationContext* y administrar es anotándolos con estereotipos. El estereotipo más básico es **@Component** y si anotamos a una clase con él, simplemente estamos diciendo que es un bean, una clase administrada por Spring y se encargará de instanciarla e inyectarla cuando sea necesario.

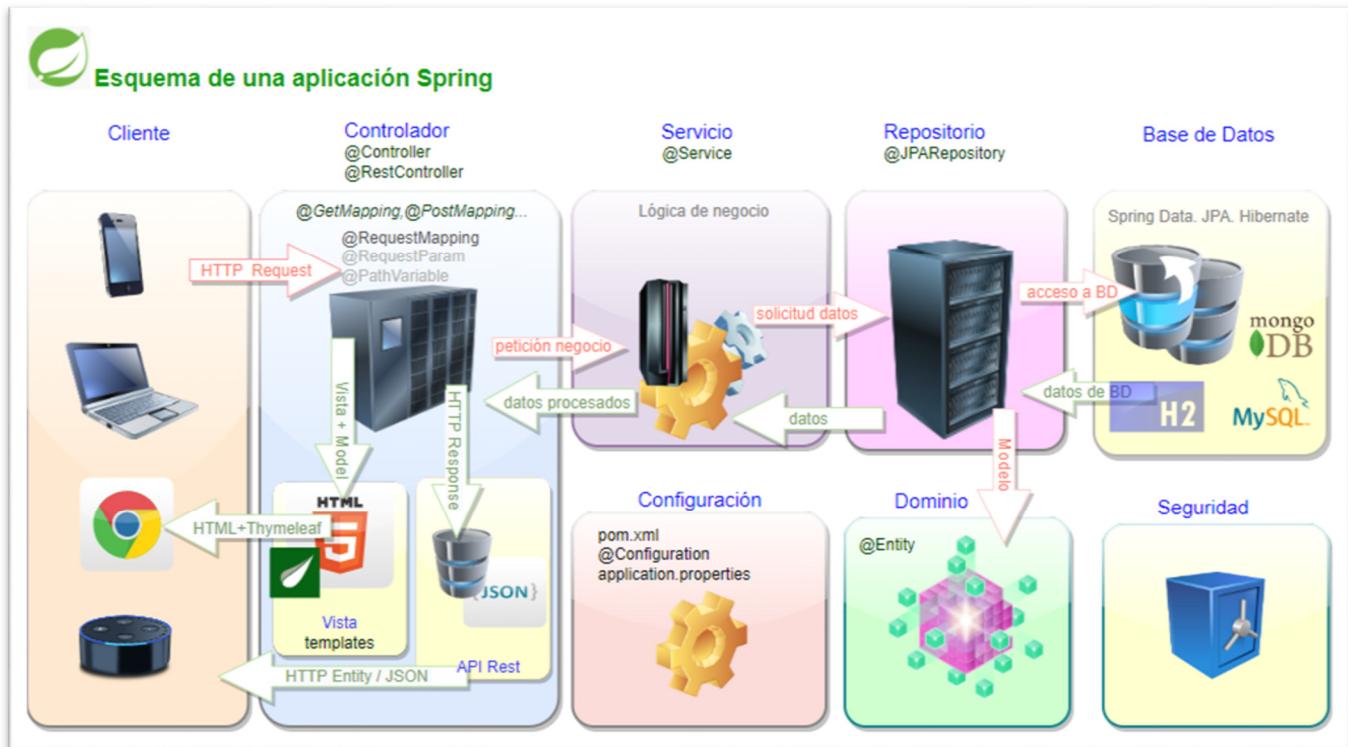
Existen estereotipos más especializados, derivados de **@Component**, que además de la funcionalidad que acabamos de mencionar, proporcionan beneficios de automatización adicional y más semántica a nuestro código. Los que usaremos en este manual serán:

@Controller: anota clases de la capa de controladores, es decir, la capa que recibe las peticiones del cliente. Esas peticiones vienen dadas por comandos HTTP como GET, POST, etc.

@Service: anota clases en la capa de servicio. Son clases que típicamente gestionan la lógica de negocio e invocadas desde los controladores. Será frecuente entonces que en una clase anotada con **@Controller** tenga “inyectado” un **@Service**; es decir que tenga un atributo de tipo *Service* que será creado por Spring para nosotros.

@Repository: anota las clases en la capa de persistencia, que actuará como repositorio de la base de datos. En muchos casos las clases anotadas con @Service hacen una solicitud de datos a las clases repositorio.

En el siguiente esquema podemos ver el flujo completo de una petición HTTP del cliente, con todas las capas y los estereotipos implicados.



@Component vs. @Bean

Es una confusión bastante habitual mezclar el concepto de los estereotipos de Spring, como @Component y la anotación @Bean.

Como comentamos en el apartado anterior, si anotamos una clase con @Component (o alguna de sus especializaciones) esa clase será detectada automáticamente mediante el escaneo de classpath y creará un nuevo bean para ella. Por eso situamos la anotación a nivel clase.

@Bean se usa para declarar explícitamente un solo bean, en lugar de dejar que Spring lo haga automáticamente como lo hicimos con @Controller. Separa la declaración del bean de la definición de la clase y le permite crear y configurar beans exactamente como deseemos. **La anotación @Bean no se coloca a nivel de clase** (si lo haces se producirá un error de compilación), **se sitúa a nivel método y será ese método el que devolverá el bean**.

@Bean

```
public NombreBean getNombreBean(){
    // código necesario para crear el bean
    // instanciaNombreBean = new NombreBean();
    return instanciaNombreBean;
}
```

En muchas ocasiones, los métodos que devuelven *beans* se declaran dentro de las clases anotadas @Configuration, de forma que se ejecuten al iniciar la aplicación.

Scopes

Por “scope” entendemos el ámbito, o forma de crear las instancias de los componentes registrados en el Application Context. Spring por defecto incluye 7 scopes diferentes: Singleton, Prototype, Request, Session, Global sesión, Application y Websocket. Vamos a describir las cuatro primeras;

Singleton: es el ámbito por defecto de los @Component, e implica que el contenedor de Spring creará una única instancia compartida para toda la aplicación, por lo que siempre que se solicite este Bean se estará inyectando el mismo objeto. La instancia se almacenará en un caché gestionado por Spring.

Prototype: implica que el contenedor de Spring creará una nueva instancia del objeto cada vez que se le haga una petición. Una de las reglas básicas que indica la documentación de Spring es que para entornos donde haya un mantenimiento de sesión (Stateful) se emplee el Prototype Scope y cuando no se mantenga la sesión del usuario (Stateless) se emplee el Singleton Scope, no obstante, puede haber más casos de uso.

Request: El contenedor de Spring creará una nueva instancia del objeto definido por el Bean cada vez que reciba un HTTP request.

Session: El contenedor de Spring creará una nueva instancia del objeto definido por el Bean para cada una de las sesiones HTTP (Aplicación Stateful) y entregará esa misma instancia cada vez que reciba una petición dentro de la misma sesión.

La forma de indicar el *scope* a un componente es añadiendo la anotación @Scope después de su estereotipo. Como ya comentamos en caso de ser Singleton no es necesario indicarlo. Ejemplo:

```
@Component  
@Scope("session")  
public class Componente { . . . }
```

Herramientas para desarrolladores

Para finalizar este capítulo vamos a hacer un pequeño resumen de las herramientas interesantes para cualquier desarrollador y que trataremos en mayor o menor medida durante este curso:

- IDE: “Entorno de Desarrollo Integrado” es una aplicación en la que desarrollaremos nuestras aplicaciones. Están diseñados para abarcar todas las tareas de programación en un mismo sitio, ofreciendo una interfaz central con todas las herramientas que necesita un desarrollador, como editor de código, compilador, depurador o herramientas de automatización. Los hay orientados a lenguajes específicos (PyCharm para Python), Android Studio (para aplicaciones Android en Java o Kotlin), XCode (para desarrollos IOS) o genéricos para usar con cualquier lenguaje (Visual Studio, IntelliJ Idea, etc). *Netbeans se situaría a medio camino ya que nació para desarrollo en Java, pero permite de forma nativa programar en otros lenguajes como PHP y dispone de plugins para añadir otros lenguajes.*
- Herramientas para control de versiones: Git. Lo podemos usar en modo consola, con el propio IDE o con herramientas gráficas como Source Tree, tanto para trabajar en local como con repositorios remotos (GitHub, GitLab).
- Utilidades para testing: JUnit, Mockito. También podemos incluir aquí *Postman*, aunque es más una herramienta que nos ayuda en el desarrollo de API Rest, más que de testing puro.
- Herramientas de autocompletado de código: además del clásico autocompletado de todo IDE como puede ser *Emmet* (disponible en Visual Studio sin necesidad de instalación adicional alguna) existen herramientas con IA como Kite y Github Copilot.

- Aplicaciones de gestión de tareas para organizar nuestro trabajo: generalmente orientadas a metodologías ágiles como Scrum, entre las más populares: Jira, Trello, Clickup, etc.
- Utilidades para la documentación: JavaDoc, Swagger para API.
- Gestores de contenedores: Docker y herramientas que faciliten el *deploy* de nuestras aplicaciones (lo que se conoce como Integración Continua): GitHub Actions, Jenkins...
- Webs con documentación oficial y foros de consultas (stackoverflow).

Videos de apoyo

Como complemento de este manual, disponemos de una lista de reproducción de videos propios que te pueden ayudar a seguir el curso:

<https://www.youtube.com/playlist?list=PLUSxu1SUrJ9h8qfwuXHpoImMGZ0967Bc5>

Además, la plataforma openwebinars, a la que tienes acceso gratuito, dispone de cursos interesantes con un enfoque similar al desarrollado en este manual.

Temas 3, 4, 5, 6 y 7: <https://openwebinars.net/academia/aprende/spring-boot/>

Tema 7: <https://openwebinars.net/academia/aprende/hibernate/>

Tema 8: <https://openwebinars.net/academia/aprende/spring-boot/>

<https://openwebinars.net/academia/aprende/seguridad-api-rest-spring-boot/>

Tema 9: <https://openwebinars.net/academia/aprende/api-rest-spring-boot/>

<https://openwebinars.net/academia/aprende/api-rest-spring-boot-avanzado/>

Estos son otros cursos interesantes de la plataforma OpenWebinars:

<https://openwebinars.net/academia/aprende/spring-framework/>

<https://openwebinars.net/academia/aprende/spring-core/>

Por último, en Youtube hay multitud de tutoriales que también pueden servir de apoyo para comprender la materia y para ver diferentes formas de enfocar una aplicación con esta tecnología. Estos son algunos ejemplos:

- https://www.youtube.com/watch?v=ws_7bgqK71w&list=PLRFOqDrY-6nsh7CmWF3LfLjCyZG-zoA-T una serie de videos que explica los conceptos básicos del framework.
- https://www.youtube.com/playlist?list=PL5leSwLYapPbUxGbzWf6cb9BLP_TEqRAR buena lista de reproducción, con videos cortos, en la que se tratan todos los aspectos del curso
- https://www.youtube.com/playlist?list=PLRFOqDrY-6nu_k7XrUglgd5SA3Vg7TiWe muestra el desarrollo de una aplicación paso a paso con Thymeleaf. Incluye el código en Github.
- <https://www.youtube.com/playlist?list=PLcal8vM1NK3ttWdID2mW9FBp-U-QoEt5K> es otra lista de reproducción bastante completa de diferentes aspectos de una aplicación con Spring.
- https://www.youtube.com/watch?v=xNZEgVJ_4Q0 se centra en la construcción de un CRUD mediante API REST y muestra como probarla con Postman.

Tema 2: Instalación y Primer Proyecto

Instalar JDK

Como ya sabemos, Java™ Development Kit (JDK) es el software imprescindible para desarrollar cualquier tipo de aplicación en Java: incluye el intérprete Java, clases Java y herramientas de desarrollo Java (JDT): compilador, depurador, generador de documentación, etc.

Podemos instalarlo de forma independiente, o bien mediante las utilidades que nos ofrece Visual Studio. Vamos a hacerlo con la primera opción, descargamos el zip de **JDK 17** desde:

<https://www.oracle.com/java/technologies/downloads/#jdk17-windows>

Descomprimimos en la misma carpeta y movemos la carpeta obtenida a *c:\Archivos de Programa\java\jdk-17.0.3.1*.

Visual Studio Code

Ya comentamos en el tema anterior lo que era un IDE. En nuestro caso usaremos Visual Studio Code, uno de los más empleados actualmente: es multiplataforma, gratuito y de código abierto, multilenguaje y es configurable y personalizable con multitud de extensiones para desarrollar de forma rápida en prácticamente cualquier lenguaje y framework. Es uno de los recomendados oficialmente por Spring para desarrollar bajo este framework.

Instalación

Para su instalación seguiremos estos pasos:

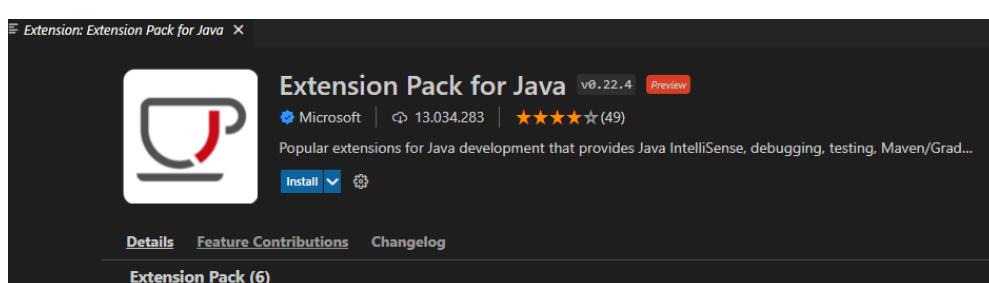
- 1.- Ir a la página de descargas de la herramienta: <https://code.visualstudio.com/#alt-downloads>
- 2.- Seleccionar Windows - System Installer 64 bits y descargar. (La versión *System Installer* lo instala en Archivos de Programa mientras que la versión *User Installer* lo instala en la carpeta del usuario)
- 3.- Iniciamos instalación haciendo doble clic sobre el archivo descargado. Aceptamos acuerdo de licencia y dejamos la carpeta de instalación por defecto: C:\Program Files\Microsoft VS Code y dejamos el resto de opciones de la instalación con la configuración ofrecida.

Configurar Visual Studio Code

- 1.- La primera vez que lo ejecutemos nos ofrecerá opciones de configuración:

- Cerramos la pestaña "Get Started" sin cambiar ningún parámetro.
- En la ventana emergente "Instalar paquete de idioma español" no lo instalaremos para acostumbrarnos a trabajar en inglés.

- 2.- El paso siguiente será instalar las Extensiones para trabajar con Java y Spring. Para ello pulsamos el  botón en la barra lateral o pulsamos Ctrl + Alt + X. Buscamos "Java Extension Pack for Java" y pulsamos **[Install]**.



3.- Buscamos "Spring Boot Extension Pack" y pulsamos *[Install]*. Este pack incluye:

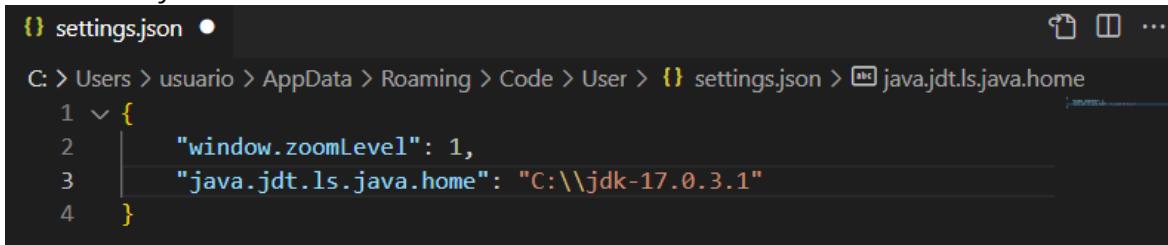
- Spring Boot Extension: herramientas básicas para editar y configurar proyectos Spring.
 - Spring Initializr: generación rápida de la estructura de un proyecto con las dependencias necesarias.
- Más información sobre VSC: <https://code.visualstudio.com/docs/?dv=win64>

4.- Configuración de JDK. Debemos indicarle la ubicación del jdk que emplearemos para compilar y ejecutar nuestros proyectos. Hay varias formas de hacerlo, una de ellas sería actualizando la variable de entorno JAVA_HOME, aunque esto podría afectar a otros programas de nuestro ordenador. Otra forma, y es la que vamos a emplear, es añadir a nuestro fichero de configuración **settings.json** la variable **java.jdt.ls.java.home** con la ruta del JDK. Esto podemos hacerlo a nivel usuario (para todos sus proyectos) o a nivel workspace (o proyecto). Para ello:

- Paleta de comandos (*Cntrl + May + P*): escribimos "Preferences: Open User Settings (JSON)"
- Añadimos la línea con la variable y su valor. Recuerda que en un archivo *json* las líneas (par variable: valor) deben estar separadas por una coma.

`"java.jdt.ls.java.home": "C:\\\\jdk-17.0.3.1"`

- Guardar el archivo y reiniciar Visual Studio Code.



The screenshot shows the Visual Studio Code interface with the settings.json file open. The file path is shown in the top left: C: > Users > usuario > AppData > Roaming > Code > User > settings.json. The file content is as follows:

```
1 <pre>{"2 <pre> 3 <pre> "window.zoomLevel": 1,4 <pre> "java.jdt.ls.java.home": "C:\\\\jdk-17.0.3.1"5 <pre> }</pre>
```

Para más información consultar:

<https://code.visualstudio.com/docs/java/java-tutorial>

En caso de tener varios JDK instalados consultar:

https://code.visualstudio.com/docs/java/java-project#_configure-runtime-for-projects

Podemos configurar muchos más parámetros. Para ello, desde la paleta de comandos, podemos escribir: *Preferences: Open User Settings* y buscar las que consideremos interesantes y configurarlas, por ejemplo, el autoguardado de archivos, formato automático al guardar, etc. Todos ellos añadirán nuevas entradas en el archivo *settings.json*.

Instalación de Maven



Apache Maven es un software que se encarga de la construcción y gestión de nuestros proyectos, haciendo especial hincapié en la gestión de dependencias, de forma que, si nuestro proyecto necesita una determinada librería externa, Maven se encarga de su descarga e incorporación al proyecto de forma sencilla (solo debemos incluir esa dependencia en el fichero de configuración pom.xml). Además, Maven dispone de comandos de consola, entre otros para la ejecución de nuestra aplicación.

Con lo que acabamos de comentar podríamos pensar que es imprescindible instalar Maven para el desarrollo de proyectos Spring, pero en realidad, Spring Boot incluye un plugin llamado "Maven Wrapper" que nos permite ejecutar las funcionalidades de este software sin necesidad de instalarlo. Veremos más adelante como nuestro proyecto SpringBoot tendrá archivos y carpetas comenzando por "mvn" que se corresponden con este plugin.

En nuestro caso, vamos a instalarlo para tener mayor flexibilidad y poder ejecutar sus comandos de consola en caso de que surjan problemas con algún proyecto. Para instalarlo:

- Descargar el zip con los binarios Apache Maven desde: <https://maven.apache.org/download.cgi> y descomprimirlo en una carpeta cualquiera. Podemos descargarlo "cerca" del JDK por tener todo bien organizado.
- En Visual Studio, instalar la extensión "Maven for Java".
- Añadir en la variable de entorno **PATH** de nuestro sistema, la carpeta **/bin** del directorio donde hemos instalado Maven y en la variable **JAVA_HOME** la ruta del jdk (sin **/bin**).
- Para comprobar que todo está ok, reiniciar y abrir una consola CMD: `mvn -version`

Una vez instalado, los comandos más habituales que emplearemos son los siguientes:

- `mvn clean` : borra la carpeta `/target` (código compilado)
- `mvn package` : compila la aplicación y genera el archivo "jar" en la carpeta `/target`
- `mvn clean package` : ejecuta los dos comandos anteriores
- `mvn clean package -DskipTests` : igual que el anterior, pero sin ejecutar los tests que tuviésemos programados.
- `mvn spring-boot:run` : ejecuta la aplicación Spring Boot.
- `mvn test` : ejecuta los test que tengamos programados

Git y GitHub

Git no forma parte de Visual Studio Code, sino que es un software independiente de control de versiones universalmente utilizado. Por otra parte, GitHub es un sitio web que permite alojar aplicaciones en la nube y permite la gestión de sus versiones con Git.

Sería muy extenso y no abordable en este manual explicar las características de Git. Aunque está basado en comandos de texto, Visual Studio Code lo tiene integrado de forma que no es necesario conocer estos comandos, sino que podemos operar directamente desde los menús del IDE.

Descarga e instalación

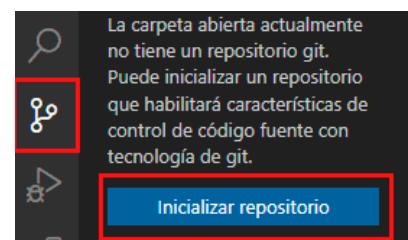
Git se descarga gratuitamente desde su página oficial: <https://git-scm.com/downloads> y su instalación es sencilla, pudiendo dejar las opciones por defecto que nos ofrece el asistente de instalación. La propia instalación añadirá a la variable PATH la ruta de la aplicación.

No es necesario instalar ninguna extensión en Visual Studio Code para usar Git, podemos acceder a todas sus operaciones desde el panel "Source Control" que se activa al pulsar el ícono lateral de control del versiones:  En la web: <https://programacionfacil.org/blog/como-utilizar-git-desde-visual-studio-code/> tienes más información sobre su uso. Existen extensiones como *Git Graph*, *Git Lens*, *Git Blame*, etc. que pueden optimizar la gestión de versiones desde VSC aunque nosotros no las usaremos.

Inicio del repositorio

En Visual Studio abrimos la carpeta raíz del proyecto del que queremos mantener versiones. Pulsamos el ícono lateral indicado previamente y se abrirá el panel: *Control de Versiones*. Debemos pulsar el botón: *Iniciar Repositorio*. A partir de este momento, esa carpeta y sus subcarpetas estarán bajo control de versiones, aunque en principio, solo a nivel local.

Esta carpeta raíz puede ser un proyecto o una carpeta base que englobe distintos proyectos.

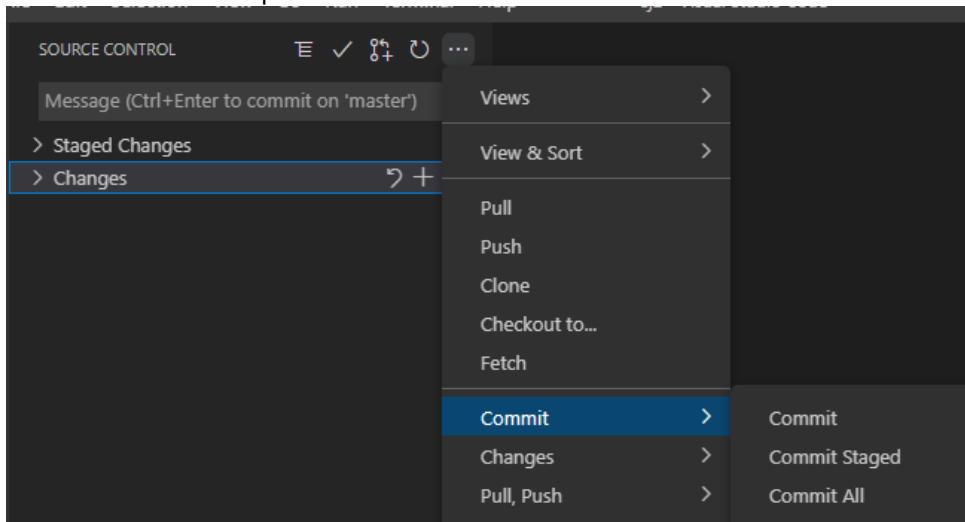


La primera vez que creamos un repositorio nos solicitará una configuración inicial consistente en nuestro email y nombre. Para asignarlo, simplemente abrimos un terminal y ejecutamos estas dos órdenes:

```
git config --global user.email "you@example.com"  
git config --global user.name "Your Name"
```

Comandos git

Como hemos comentado no vamos a explicar la arquitectura de git y sus comandos, simplemente comentar que todas las operaciones que realizaremos estarán disponibles en el panel *Control de Versiones*, en el ícono de los tres puntitos.



La operación más habitual será hacer **commit** de los cambios producidos y no es necesario hacer *stage* previamente, ya que se pueden hacer ambas operaciones conjuntamente. En el panel de explorador, los archivos modificados a los que no se les haya hecho commit, aparecerán en distintos colores y con una letra indicando su situación (untracked, staged...).

Sincronización con Github

Si crear el repositorio remoto en GitHub, desde el panel *Control de Versiones*, con la opción de menú "*Publish*" nos pedirá nuestras credenciales en Github para, a continuación, crear un nuevo repositorio remoto en GitHub con el mismo nombre que el local (nos preguntará si lo queremos hacer público o privado).

Otra opción, para que no cree el nuevo repositorio en GitHub, es crearlo nosotros previamente "a mano" desde la página de Github, copiar la URL asignada y en VSC, añadirlo mediante la opción: **Remote > Add Remote**. Si trabajamos en varios ordenadores, y ya hemos creado el repositorio en Github anteriormente, esta será la opción a emplear en los nuevos ordenadores en los que trabajemos.

Si queremos actualizar al revés, desde GitHub a local (por ejemplo, para tener en el ordenador de casa todo el trabajo realizado en clase y subido a GitHub), utilizaremos la opción *Pull* del panel.

Permisos en GitHub

Sobre un repositorio, sea público o privado, se pueden añadir colaboradores que tengan acceso al mismo. En la página de GitHub: *Settings > Collaborators*. Para ver todos los repositorios de los que eres colaborador: <https://github.com/> (sin nada más en la URL). Por el contrario, para dejar de ser colaborador, en tu propio perfil > *Settings > Repositories* > "Leave" en el repositorio seleccionado.

Más sobre Visual Studio Code

Teclas aceleradoras más útiles:

- **Cntrl + May + P**: Paleta de comandos.
- **Cntrl + +** (o bien **-**): Zoom in (o bien zoom out).
- **Cntrl + F5**: Ejecutar aplicación (sobre el archivo con el main). Al lanzarlo nos aparece la ventana de herramientas para gestionar el servidor:



- **Cntrl + C** y **Cntrl + V**: duplica línea en la que está el cursor, sin tener que seleccionarla
- En documentos HTML, nombre de etiqueta sin <,> + TAB: genera la etiqueta completa (*doc + TAB genera un esqueleto de documento completo*)
- **Alt + ↑ Flecha arriba / Alt + ↓**: Mueve arriba/abajo la línea en la que está el cursor. *Si hay un grupo de líneas seleccionada, mueve todas.*
- **Cntrl + K C**: Comenta la línea actual, o el bloque seleccionado.
- **Cntrl + K U**: Descomenta la línea actual, o el bloque seleccionado.
- **Cntrl + S**: Guardar documento.
- **May + Alt + F**: Formatea el documento para que quede bien indentado.
- **May + Alt + O**: Añade los imports necesarios de las clases empleadas en el archivo.
- **F2**: Renombrar archivos, variables, etc.
- **F8**: Si hay errores de compilación, nos sitúa en ellos (menú superior: *Go > Next Problem*)

Extensiones interesantes:

- Thymeleaf (tiene snippets de código)
- HTML preview (permite ver la apariencia HTML en el propio VSC, icono arriba derecha)
- Open in browser (con botón derecho puedes mostrar la página en el navegador que quieras).
- MySQL (cliente para MySQL), DataBase client for VS Code, DataBase Explorer
- Thunder Client (como Postman, pero sin necesidad de registro ni conexión a internet)
- Snippets de código Java y Spring
- SpotBugs, CheckStyle, etc.

Configuración:

En el archivo **settings.json** (*donde añadimos la variable java.jdt.ls.java.home*) podemos añadir otros parámetros de personalización, por ejemplo, si no queremos que nos escriba el nombre de los parámetros de los métodos, añadiríamos: **"java.inlayHints.parameterNames.enabled": "none"**

Otro parámetro que podemos incluir es la gama de colores, por ejemplo, para fondo claro:

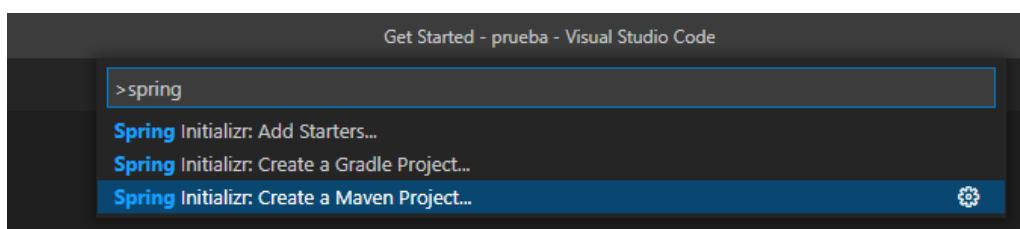
"workbench.colorTheme": "Solarized Light" o bien **"Default Light+"** aunque también podría cambiarlo desde la paleta de comandos: *Preferences >> Color Theme*.

Primer proyecto

Tenemos dos formas de crear un proyecto desde cero, o bien a través de la web oficial que Spring nos ofrece para ello: <https://start.spring.io> o bien a través del propio Visual Studio. Si elegimos la primera opción nos descargará un zip que debemos descomprimir sobre una carpeta y en el segundo caso nos pedirá el nombre de la carpeta “padre” sobre la que creará la carpeta del proyecto.

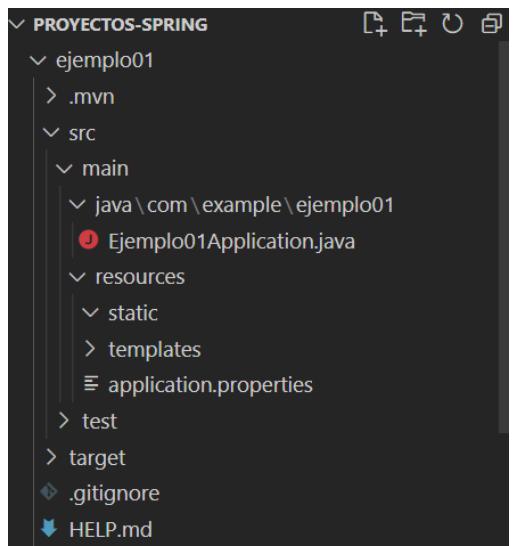
VSC trabaja con el concepto de *WorkSpace* que podemos asimilar a carpeta de proyecto. Podríamos crear subcarpetas con distintos proyectos relacionados, pero en ciertos casos pueden provocar algún conflicto. Todas las subcarpetas del workspace se mostrarán en un solo panel lateral de VSC.

Para crear el proyecto desde VSC, abrimos la paleta de comandos (*Menú View > Command Palette o bien Cntrl + May + P*) y seleccionamos **“Spring Initializr: Create a Maven Project”**



Seguimos los pasos que nos va indicando:

- **Versión de Spring:** dejamos la que nos indica por defecto
- **Lenguaje:** Java
- **Group id:** esta será la raíz de los paquetes de nuestro proyecto, suele ser *com.nombreOrganizacion*, pero por ahora podemos dejarlo con su valor por defecto.
- **Artifact id:** será el nombre del programa, podemos generar un estándar de nombres para los ejercicios de este curso, pero en este primer caso, le podemos dejarle su nombre por defecto.
- **Packaging type:** jar. Al generar un archivo jar, el propio programa tendrá un servidor web embebido por lo que podremos ejecutarlo de forma independiente. El empaquetado "war" sería para ubicar el proyecto en un servidor web externo.
- **Versión de Java:** Deberá coincidir con alguno de los jdk que tengamos instalado, vamos a seleccionar 17, que es la última versión de Java LTS (Long Time Support)
- **Dependencias:** marcaremos aquí las dependencias (librerías) que serán necesarias para nuestro proyecto. Por ahora solo seleccionaremos starter-Web, Spring Boot DevTools, aunque más adelante incorporaremos otras como Lombok o las relativas al gestor de base de datos que empleemos en nuestros proyectos. Pulsamos *[Enter]* cuando no queramos introducir más.



Finalmente, nos pregunta por la carpeta en la que guardar el proyecto, seleccionamos una carpeta raíz en la que almacenaremos todos nuestros proyectos. Visual Studio creará una carpeta con el nombre que le hayamos dado a nuestro artifact id, con la estructura de archivos y carpetas de un proyecto Spring.

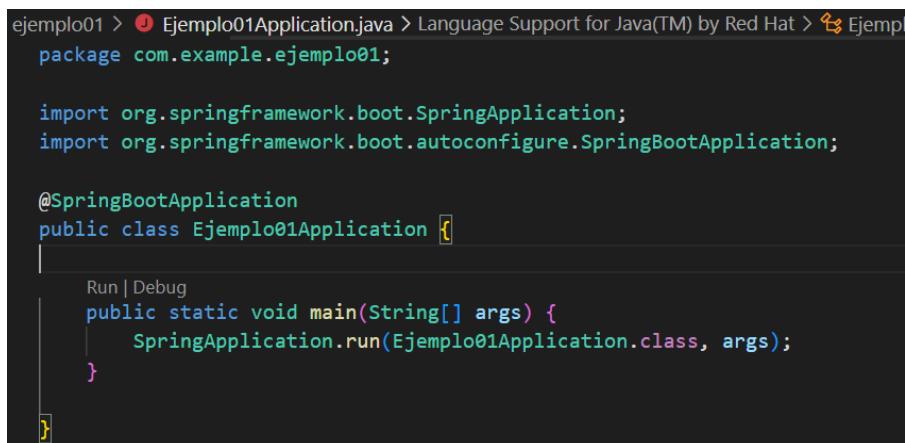
Estructura de un proyecto SpringBoot

En el panel lateral "Explorer" podemos ver la estructura del proyecto creado. En la imagen siguiente tendríamos una carpeta raíz llamada en este caso *PROYECTOS-SPRING* y en ella se habría generado la carpeta 'ejemplo01' al crear el proyecto. Podemos ver nuestro proyecto desde ese panel o desde el panel **Java Projects**.

Viendo la estructura del proyecto vemos las carpetas generadas, siendo 'src' donde estará todo el código fuente y la que más nos interesa en estos momentos.

- **src\main** contendrá, organizado en paquetes, el código fuente de nuestra aplicación.
- **resources\static** contendrá elementos estáticos como imágenes, archivos HTML (estáticos), CSS, etc.
- **resources\templates** contendrá como veremos más adelante las páginas HTML que verán nuestros visitantes, pero HTML que contendrá código dinámico.
- Fichero **application.properties** que mantiene configuración por defecto del proyecto y al que podemos añadir parámetros de comportamiento. Por ejemplo, se le puede cambiar el puerto en el que se ejecutará el servidor embebido. Los cambios en el fichero de propiedades implican recargar el proyecto.

En la carpeta **src/** vemos que ya hay un primer archivo .java que contiene el método *main* y será lo que arranque primero el servidor web (por defecto Tomcat) y a continuación nuestra aplicación. Esta clase deberá estar anotada con **@SpringBootApplication**.



```

ejemplo01 > Ejemplo01Application.java > Language Support for Java(TM) by Red Hat > Ejemplo01Application.java
package com.example.ejemplo01;

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

@SpringBootApplication
public class Ejemplo01Application {

    public static void main(String[] args) {
        SpringApplication.run(Ejemplo01Application.class, args);
    }

}

```

Otro panel interesante es el panel “Java Projects”, aparece en cuando clicamos en un archivo java (en la carpeta src/main). Desde este panel crearemos nuevos paquetes en nuestro proyecto.

pom.xml y dependencias starter

Uno de los ficheros más importantes dentro de un proyecto SpringBoot con Maven es el *pom.xml* en el que se refleja la configuración que le asignamos al crear el proyecto y que podremos modificar, por ejemplo, para añadirle nuevas dependencias, cambiar el empaquetado de *jar* a *war*, cambiar la versión de Java, etc.

Nuestro *pom.xml* (**pom simple**) hereda configuración por defecto de un pom “padre”, o **super pom** por lo que el *pom.xml* real (se llama **pom efectivo**) es más completo aún del que vemos en el proyecto. Así nos despreocupamos de mucha de la configuración.

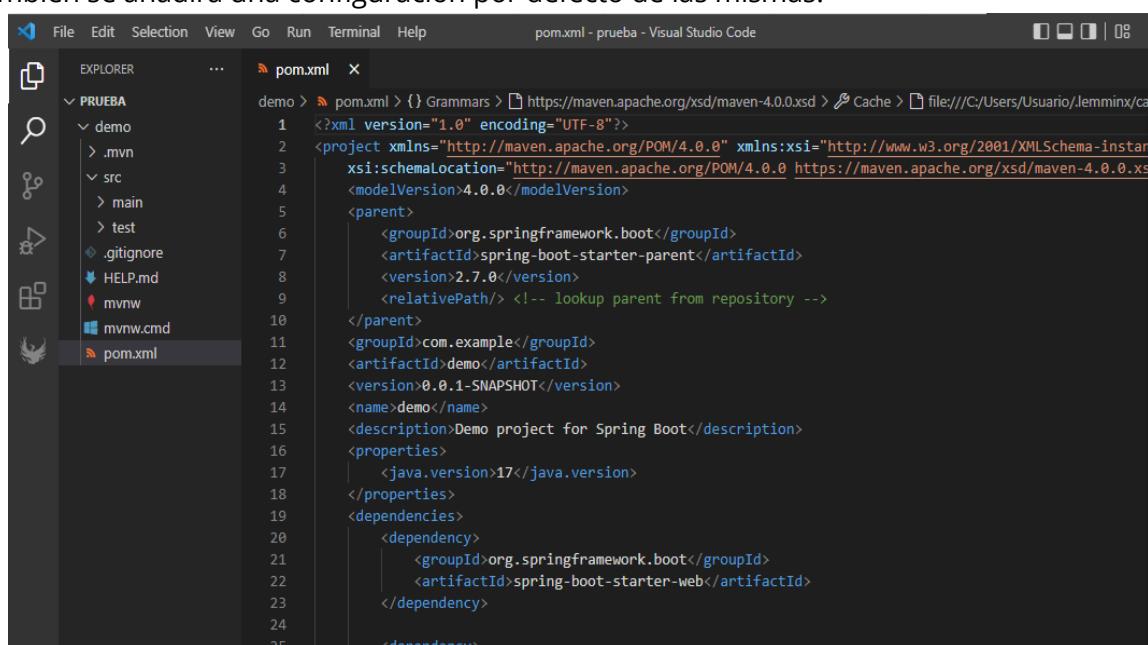
```

<parent>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-parent</artifactId>
    <version>3.1.5</version>          <!-- Versión octubre 2023: ver en initializr la actual -->
    <relativePath/>                <!-- Lookup parent from repository -->
</parent>

```

Otra cosa que observamos es que las dependencias que llevan el prefijo *starter* no incluyen la versión. También es gestionado de forma automática mediante el pom “padre”.

Las dependencias *starter* son unas dependencias (librerías necesarias para nuestra aplicación) que añadirán a su vez otras dependencias necesarias para el funcionamiento de las primeras, y así transitivamente. Esto hará que se añadan un número elevado de jar, pero lo más importante es que también se añadirá una configuración por defecto de las mismas.



```

<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
          xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 https://maven.apache.org/xsd/maven-4.0.0.xsd"
          modelVersion="4.0.0">
    <parent>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-parent</artifactId>
        <version>2.7.0</version>          <!-- lookup parent from repository -->
        <relativePath/>                <!-- lookup parent from repository -->
    </parent>
    <groupId>com.example</groupId>
    <artifactId>demo</artifactId>
    <version>0.0.1-SNAPSHOT</version>
    <name>Demo</name>
    <description>Demo project for Spring Boot</description>
    <properties>
        <java.version>17</java.version>
    </properties>
    <dependencies>
        <dependency>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-starter-web</artifactId>
        </dependency>
    </dependencies>

```

Así pues, las dependencias starter nos quita mucho trabajo de elegir las dependencias necesarias, qué otras dependencias necesitan estas, versiones de cada una de ellas y su configuración. Por ejemplo, si añadimos la dependencia starter "jpa" añadirá todas las dependencias para trabajar con repositorios JPA como veremos más adelante.

Además de añadir las dependencias starter en el momento de crear el proyecto como acabamos de ver, también podríamos añadirlas posteriormente, durante el desarrollo del proyecto. Para ello, debemos buscarlas en el repositorio Maven (mvnrepository.com) y añadirlas directamente en el pom.xml

mvnrepository.com/search?q=spring+boot+starter

MVNREPOSITORY

spring boot starter

Found 27388 results

Sort: relevance | popular | newest

1. Spring Boot Starter Test
org.springframework.boot » spring-boot-starter-test
9,774 usages Apache
Starter for testing Spring Boot applications with libraries including JUnit Jupiter, Hamcrest and Mockito
Last Release on May 19, 2022

2. Spring Boot Starter Web
org.springframework.boot » spring-boot-starter-web
9,035 usages Apache
Starter for building web, including RESTful, applications using Spring MVC. Uses Tomcat as the default embedded container

Todas las oficiales empiezan por "spring-boot-starter-" y las más usuales son:

- **web**: para aplicaciones Web, API REST, MVC.
- **data-jpa**: Acceso a datos mediante JPA e Hibernate
- **thymeleaf**: motor de plantillas para hacer dinámicas las vistas MVC.
- **validation**: contiene anotaciones para validaciones automáticas de atributos de las clases.
- **test**: testing con JUnit, Hamcrest y Mockito.

Serán las típicas que incluiremos en nuestros proyectos, además de **webtools**, que no es *starter*, pero permite el *reload* automático del servidor embebido cuando guardamos los cambios en el proyecto. Así no es necesario reiniciar el servidor cada vez que hacemos un cambio, esta característica es muy útil en fases de desarrollo. **Lombok** será otra dependencia frecuente de la que hablaremos más adelante.

Ejecutar un proyecto Spring

La forma más sencilla de ejecutar nuestra aplicación desde VSC será pulsando **Ctrl + F5** sobre el archivo que contiene el main (o botón derecho > *Run*). Esto realizará la compilación y, si no hay errores, lanzará el servidor web embebido (Tomcat) y comenzará la ejecución de nuestra aplicación. Con **May + F5** paramos el servidor.

Como comentábamos previamente, al ejecutarlo se nos abre un pequeño panel de gestión del servidor que nos permite pararlo, relanzarlo, etc.



En la ventana de **terminal** podemos comprobar el resultado del proceso de compilación, el lanzamiento del servidor, veremos también el puerto en el que se está ejecutando nuestra aplicación y los eventos de *logging* que se produzcan. También se mostrarán los mensajes de nuestra aplicación que hagamos mediante *System.out.print* y similares.

Para ejecutar nuestra aplicación abriremos un navegador en <http://localhost:8080>.

Podemos cambiar este puerto por defecto en el fichero *application.properties* como veremos a continuación. Como no tenemos ninguna página de inicio en nuestra aplicación, se producirá el siguiente error:



Para probar, puedes meter en la carpeta resources/static un archivo llamado index.html con un contenido cualquiera, y lo mostrará, en vez de ese mensaje de error.

Otras formas de ejecutar un proyecto, sería desde el terminal, con comandos Maven, situados en la carpeta raíz del proyecto:

```
mvn clean  
mvn install  
mvn spring-boot:run
```

También podríamos generar el archivo .jar con el comando Maven: `mvn package` (lo crearía en la carpeta target) y luego ejecutarlo desde esa misma carpeta, mediante el comando: `java -jar nombreProyecto.jar`

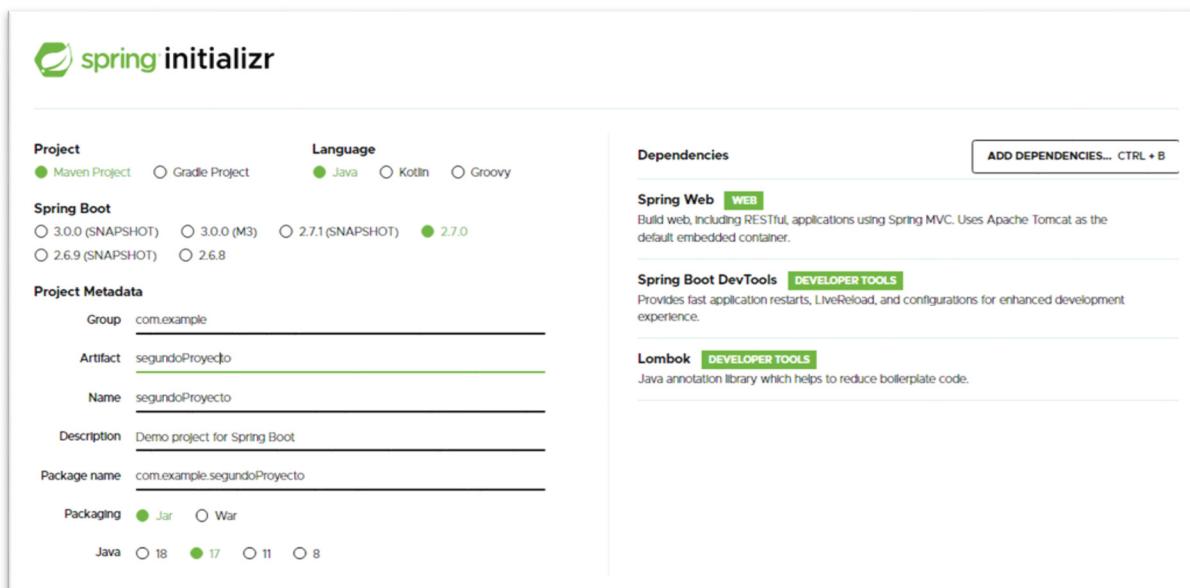


Para el comando `java -jar` utilizará la primera versión de `jdk` que se encuentre en las rutas de la variable de entorno `PATH` del sistema.

Dentro de los archivos que componen nuestro proyecto, disponemos de un archivo **mvnw.cmd** que es un script Maven Wrapper para Windows que verifica si la versión correcta de Maven está disponible. Si no lo está, descarga e instala la versión especificada de Maven para el proyecto. Podemos usar todos los comandos Maven vistos previamente por ejemplo, así: `./mvnw.cmd clean install`

SpringBoot Initializr

Otra forma de crear un proyecto, independiente del IDE con el que trabajemos, es a través de la página que nos ofrece Spring para esta tarea: <https://start.spring.io/>. En ella, parametrizaremos el proyecto al igual que hicimos desde VSC (versión de Java, artifact Id, etc) y nos generará un zip que podremos descomprimir en nuestro workspace.



Una tercera forma de crear un proyecto será **copiando y pegando otro proyecto** (la carpeta global que contiene todo el proyecto). En este curso será muy habitual, ya que haremos en los distintos ejercicios distintas versiones de una misma aplicación. Esto nos obliga a cambiar en el `pom.xml` el `ArtifactId` y el `name` y también las dependencias, en caso de que necesitásemos otras distintas. Por otra parte, el nombre por defecto asignado al paquete raíz es el del `ArtifactId`, así que, si no queremos tener que renombrarlo cada vez que hagamos una nueva copia del proyecto, deberíamos asignarle un nombre más general a ese paquete, y no el nombre del proyecto. Y lo mismo con el nombre de la clase java que contiene el `main`.

Estructura del código

SpringBoot no nos condiciona la estructura de un proyecto, pero **debemos crear todas las clases en el paquete raíz** (es el que se ha generado al crear el proyecto y donde está la clase anotada con `@SpringBootApplication`) o bien en **subpaquetes del mismo**. No podemos crear clases sin paquete (paquete "default") o fuera de esa jerarquía que parte en el paquete principal, si no, no serán reconocidas por Spring en ese escaneo de componentes que realiza el Inversor de Control que comentábamos en el primer capítulo. *¡Esto suele ser un error frecuente!*

La nomenclatura de los paquetes está pensada como un dominio web al revés, por ejemplo: `com.miempresa.miproyecto`. Ahí estará la clase con la anotación principal de nuestra aplicación: **`@SpringBootApplication`** y a partir de ahí, diferentes paquetes.

Configuración del proyecto

Existen varias formas de configurar el proyecto y todas son complementarias.

- **Archivo pom.xml:** Como ya hemos visto, en este archivo se configuran parámetros generales, como versión de JDK, empaquetado jar o war, dependencias, etc.
- **Clases con la anotación @Configuration:** Serían clases Java "normales" pero dedicadas a la configuración. **Se ejecutan al iniciar la aplicación.**
- **Archivo application.properties:** Podemos fijar ciertos parámetros en este archivo que se genera vacío cuando creamos el proyecto. Un parámetro típico es el cambio de puerto en el que se ejecutará la aplicación ya que, por defecto, se usa el puerto 8080 y este puerto suele ser empleado por muchas aplicaciones y puede generar algún conflicto a la hora de ejecutar el proyecto. Haríamos: `server.port=9000`
- **Anotaciones:** A lo largo de este manual veremos cómo al anotar clases o métodos logramos que se configuren con un comportamiento determinado. Ya hemos visto `@SpringBootApplication` que generaba una configuración completa para que nuestro proyecto funcionase. Otros ejemplos son los estereotipos vistos en el primer capítulo: `@Component`, `@Service`, `@GetMapping`....

Contenido estático

Una vez que creamos un proyecto, SpringBoot considera la carpeta `src/main/resources/static` como carpetas raíz de nuestro proyecto de cara a mapear URL de recursos. La carpeta `/static` es la ubicación habitual para el contenido estático: esto es páginas HTML estáticas, archivos CSS, imágenes, etc. Si ponemos en `/static` un fichero `index.html`, la aplicación arrancaría directamente mostrando ese documento.

Podemos crear dentro de `/static` carpetas `/img`, `/css`, etc. y en nuestros proyectos referenciarlos como si `/static` fuera la raíz de documentos. Por ejemplo, si en la carpeta `/static/img` añadimos una imagen con nombre `logo.png` la URL `http://localhost/img/logo.png` sería correcta.

O bien, en cualquier página de proyecto podríamos incluir: ``

WebJars

JQuery, BootStrap y otras librerías son recursos estáticos que se emplean frecuentemente. La forma de incorporarlos a nuestros proyectos en muchos casos es incluyendo las URL originales de las librerías (CDN) o bien descargándolas en nuestro servidor, y emplear URL locales.

Esta forma de trabajar tiene algunos problemas: en el primer caso (CDN) hay una penalización temporal en la descarga de las librerías y dependemos de que el servidor de las mismas esté operativo. En el segundo caso, nos obliga a estar descargando esos archivos y gestionar sus versiones.

Spring nos aporta mediante *WebJars* una forma de trabajar con estos recursos estáticos similar al resto de dependencias Maven, de forma que, si en nuestra configuración cambiamos la versión del recurso, se descargue de forma automática esa nueva versión de forma transparente para nosotros.

En su web: <https://www.webjars.org> podemos acceder a esas dependencias que incluiríamos en nuestro pom.xml. La web dispone de una casilla de búsqueda en la que podemos seleccionar el formato de la dependencia, en nuestro caso "Maven". Para BootStrap:

```
<dependency>
  <groupId>org.webjars</groupId>
  <artifactId>bootstrap</artifactId>
  <version>5.1.2</version>
</dependency>
```

Podemos añadir una dependencia adicional desde el repositorio Maven para no tener que indicar en cada plantilla la versión de BootStrap que estamos usando. Simplemente cambiando la versión de la dependencia anterior en el pom.xml, se actualizaría automáticamente la versión de todas las plantillas. Se dice que estamos trabajando con “versiones agnósticas”. Esta dependencia se llama *webjars-locator* y la incluimos así en el pom.xml:

```
<dependency>
  <groupId>org.webjars</groupId>
  <artifactId>webjars-locator</artifactId>
  <version>0.45</version>
</dependency>
```

Ahora tenemos que cambiar nuestras plantillas, para que “apunten” a las webjars en vez de a las URL habituales:

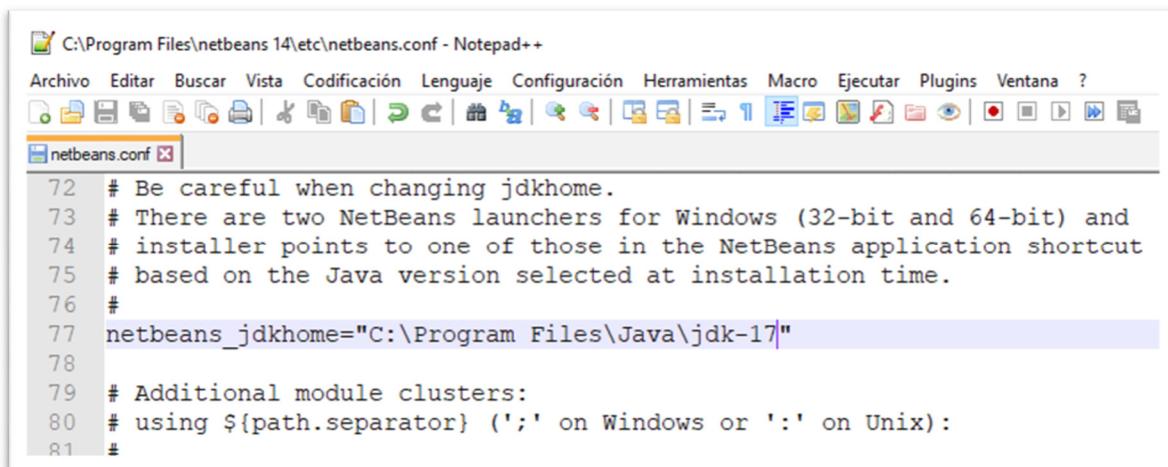
```
<link href="/webjars/bootstrap/css/bootstrap.min.css" rel="stylesheet">
<script src="/webjars/bootstrap/js/bootstrap.bundle.min.js"></script>
```

Se puede comprobar en las URL que, al incluir la dependencia *webjar-locator*, no tenemos que indicar en cada página la versión de las librerías, solo cambiando el pom.xml se actualizarían todas las páginas.

Apache Netbeans

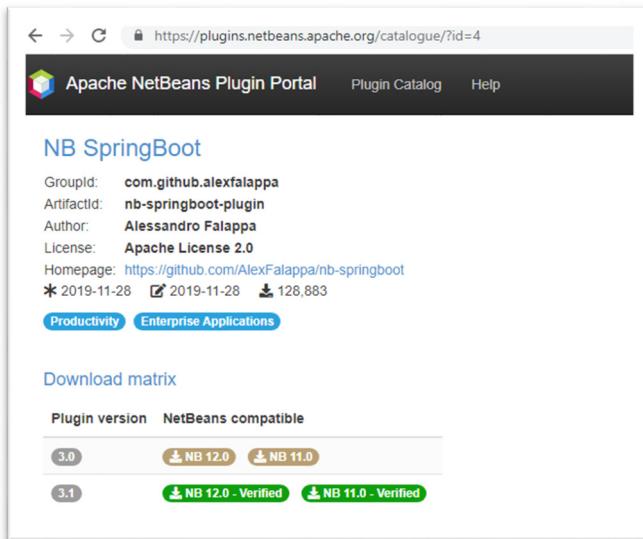
Aunque en este curso vamos a usar Visual Studio Code, también podríamos usar Netbeans para nuestros proyectos con Spring. Los pasos serían los siguientes:

- 1) Instalación de JDK igual al proceso que vimos para Visual Studio.
- 2) Instalación de Netbeans consiste en la descompresión del archivo .zip descargado de la web oficial <https://netbeans.apache.org/download/nb14/nb14.html>. En Windows podemos descomprimirlo en *c:\Archivos de Programa*. Se creará una carpeta llamada *netbeans*, que si queremos podemos renombrar a *netbeans14* o lo que queramos.
- 3) Antes de la primera ejecución hay que informar a Netbeans de dónde está instalado Java. Para ello debemos editar el archivo *netbeans.conf* situado en la carpeta *etc*, en la ruta donde esté instalado el jdk de Java (otra opción sería configurar la variable *PATH* dentro de las variables de entorno de nuestro sistema Windows para incorporar la ruta de los archivos de Java: *jdk/bin*).

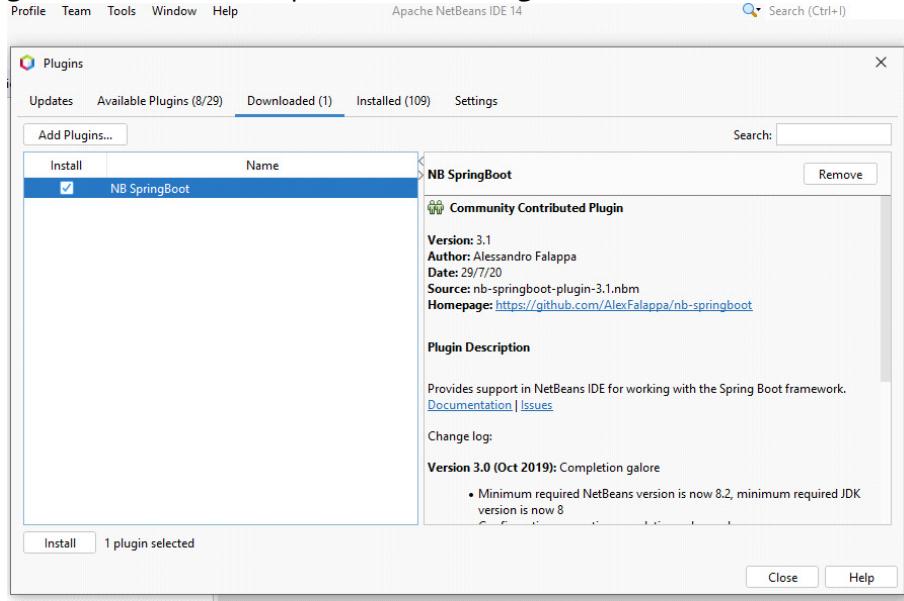


- 4) Opcionalmente, podemos crear en el escritorio un ícono del ejecutable situado en *c:\Archivos de Programa\netbeans\bin* para acceder al IDE de una forma más cómoda.

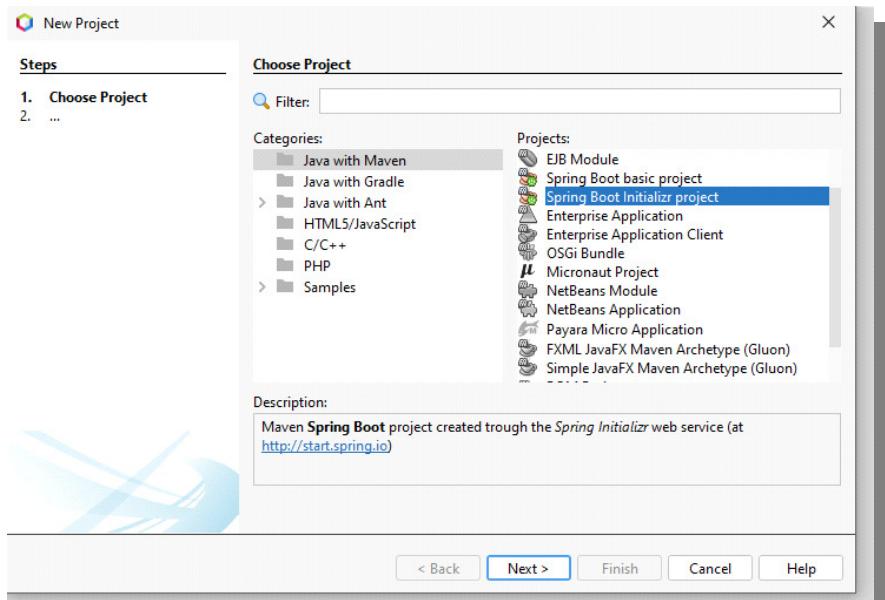
- 5) Descargar el plugin de SpringBoot desde el portal de Plugins de Apache Netbeans.



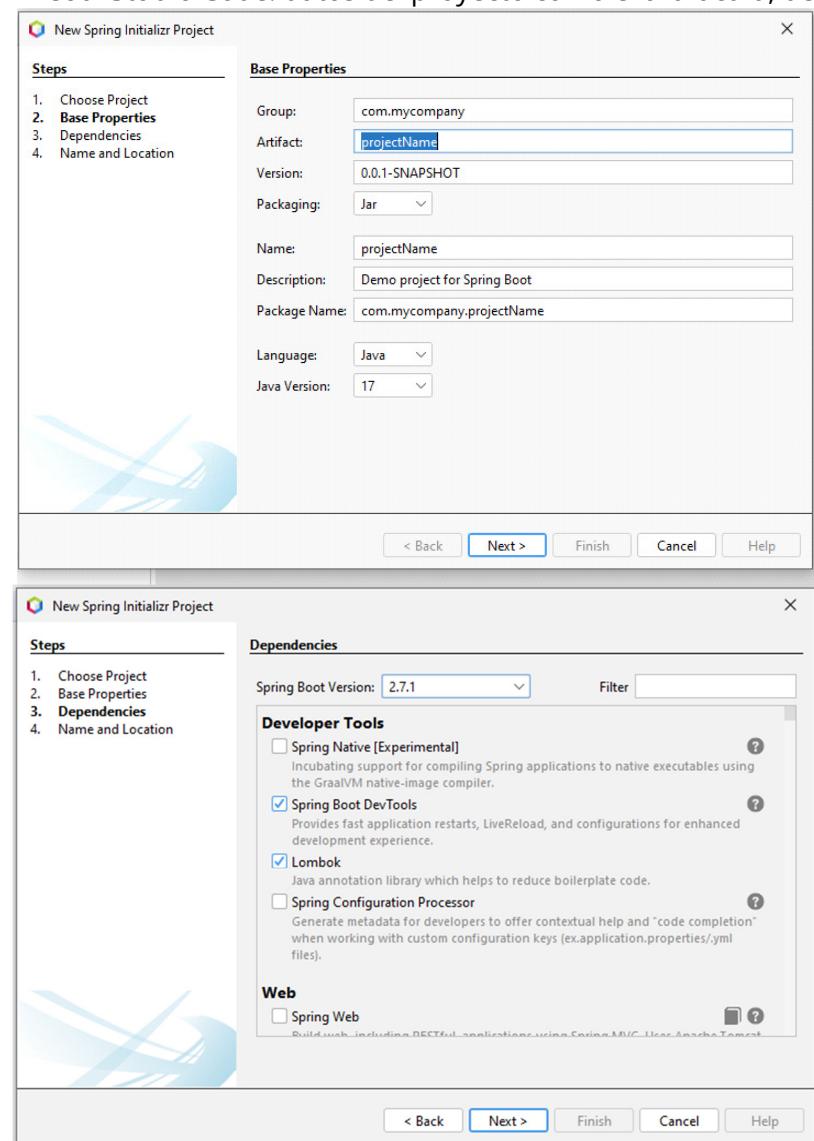
- 6) Instalar el plugin, desde el menú superior *Tools > Plugins*.



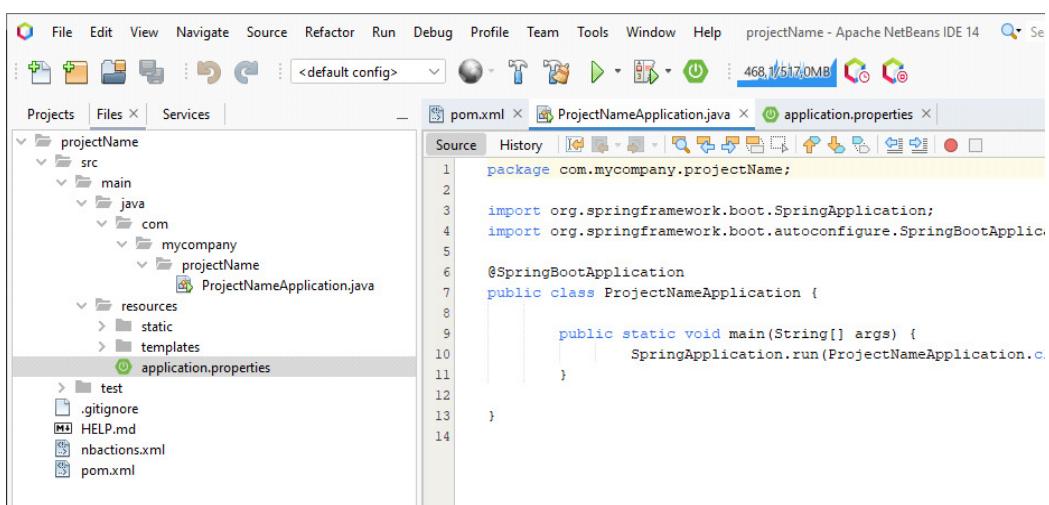
- 7) Ahora para crear un proyecto Spring Boot, menú *File > New Project. Java with Maven > Spring Boot Initializr Project*.



- 8) Luego, seguimos el asistente con los pasos de configuración del proyecto, de forma análoga al proceso visto con Visual Studio Code: datos del proyecto como el *artifact Id*, dependencias, etc.



- 9) Una vez creado, podemos ver la estructura del proyecto generado:



Tema 3: Controladores y Vistas

El controlador es la clase que se encarga de recibir las peticiones de nuestra aplicación. Las recibirán mediante el protocolo HTTP, en unos casos será de una página web creada por nosotros mismos (que será lo primero que veamos en este manual) pero también pueden venir de otro tipo de dispositivos (IoT, aplicaciones móviles, etc.). Esto lo veremos en el capítulo de API Rest.

Para crear en Spring un Controlador debemos crear una clase simple, lo que se denomina POJO (Plain Old Java Object) y anotarla con `@Controller`.

Lo siguiente que necesitaremos es crear en esa clase distintos métodos y asociarlos a las distintas peticiones HTTP (URLs) que vamos a recibir y a las que el controlador va a responder. Recordemos que mediante HTTP podemos recibir multitud de comandos, pero los más típicos son Get, Post, Put o Delete.

Entonces, cada método del controlador deberá saber a qué URL y el comando tiene que responder. Para ello, anotaremos al controlador con: `@RequestMapping` pasándole como parámetro la URL y el comando, o bien, desde la versión 4.3 Spring, anotándolo con sus derivadas `@GetMapping`, `@PostMapping`, `@PutMapping`, `@DeleteMapping`, etc. en las que el comando HTTP va implícito en la anotación y solo requieren la URL.

Características del controlador

El **nombre** que le demos a la clase controlador es indiferente. Suele ponerse nombre según su función: es típico ponerle nombres como `MainController` o `FrontController` para controladores que reciban todas las peticiones de la aplicación, o bien tener un controlador por cada área de negocio: `UserController`, `ProductController`, etc. También podemos hacer un controlador separado para la parte pública y llamarle algo como `PublicController`.

En cuanto a su **ubicación**, como decíamos previamente, **debe estar en el paquete raíz de la aplicación SpringBoot o bien en un subpaquete del mismo**. Al paquete podemos llamarle como queramos. Podemos agrupar todas las clases controladoras en un único paquete y llamarle `Controllers` o bien crear un paquete por área de negocio (Usuario, Producto, etc.) y que en ese paquete metamos el o los controladores asociados a esa área de negocio junto con otras clases que veremos en próximos capítulos, como serán los servicios o repositorios, etc.

La clase contendrá un **método** por cada comando HTTP y URL o URLs que requieran el mismo tratamiento. El interior el método contendrá las operaciones que será necesario hacer en el servidor para obtener los datos dinámicos que compondrán la respuesta y finalmente devolverá al solicitante (por ejemplo, al navegador) la respuesta.

Esas *"operaciones que se será necesario hacer en el servidor para obtener los datos dinámicos que compondrán la respuesta"* serán la lógica de negocio y pueden incluir multitud de operaciones (lógicas, matemáticas, acceso a datos, etc.) Veremos más adelante que **si estructuramos bien nuestra aplicación en capas, estas tareas (la lógica de negocio) las pasaremos a otras clases llamadas servicios y el controlador simplemente llamará a esos servicios**.

En cuanto a la respuesta, en estos primeros capítulos, será simplemente una página web con contenido estático (HTML, CSS, etc) y con contenido dinámico generado por nuestra aplicación. Estas páginas web con contenido dinámico las llamaremos **vistas**, de acuerdo con el patrón MVC.

Con un ejemplo, van a quedar todos estos conceptos más claros:

```

package com.example.myproject.controllers; //paquete para Los controladores (opcional)

import org.springframework.stereotype.Controller;
import org.springframework.web.bind.annotation.GetMapping;

@Controller
public class HomeController {
    @GetMapping("/")
    public String showHome() {
        return "index";
    }
}

```

Recuerda que **ALT** + **May** + **O** hace los imports necesarios.

En este ejemplo vemos como la anotación @GetMapping incluye como parámetro la URL que hará que se ejecute este controlador, en este caso la URL "/" sería la correspondiente a la raíz de nuestra web: <http://localhost/>

El nombre del método asociado al "mapping" puede ser cualquiera y no tiene porqué recibir ningún parámetro obligatoriamente.

El método puede devolver diferentes tipos de datos, el más sencillo es el que estamos viendo aquí, que devuelva un String. Ese String se corresponderá con la página web (la vista) que devolverá al usuario. Por defecto se sobreentiende la extensión .html para las vistas, por lo que el controlador del ejemplo devolverá la vista index.html (ojo: **debemos tener la dependencia thymeleaf en nuestro pom.xml**).

El último paso para que este ejemplo funcione, es crear la vista index.html. Esta **vista deberá estar ubicada en la carpeta *templates*** de nuestra aplicación (no en static como veíamos en el capítulo anterior) y puede ser algo tan sencillo como:

```

<!DOCTYPE html>
<html lang="es">
<head>
    <meta charset="UTF-8">
</head>
<body>
    <h1>Bienvenido</h1>
</body>
</html>

```

Obviamente, esta sería una página totalmente estática. En los apartados siguientes, mediante Thymeleaf veremos cómo añadir contenido dinámico a la vista.

Consideraciones sobre los Controladores

- **La URL que figura en la anotación es la ruta de la petición, no representa la ruta de la vista que devolverá.** Las rutas de URL de las anotaciones se estructuran en "carpetas" solo por claridad de código, pero no hay una carpeta real en el servidor. Por ejemplo: @PostMapping("/producto/alta") no implica que haya una carpeta *producto*.

- **En la anotación podemos incluir varias rutas**, por ejemplo, si con la URL <http://localhost/home> también queremos mostrar misma vista que vista con <http://localhost/>, o <http://localhost> modificaríamos la anotación añadiendo las rutas entre llaves {}, separadas por comas. Ejemplo: `@GetMapping({ "/", "/home", "" })`

Vemos entonces que no hay relación entre las URL y las páginas (vistas) que se muestran. Esto no es así en una web tradicional en HTML, en la que sí hay una relación directa entre URL y página HTML mostrada.

- Decíamos que por defecto la vista tiene que tener extensión HTML y ya no se incluye `.htm`/en el String devuelto, pero se puede cambiar esta configuración por defecto para que devuelva otro tipo de archivos, como por ejemplo los clásicos `.jsp`.

- Trabajando con SpringBoot y así evitando engorrosa configuración, las vistas deberán estar en la carpeta `templates`, pero pueden estructurarse en subcarpetas, pero habrá que reflejar esa estructura en el return del controlador. Ejemplo: `return "usuarios/bajaView"`, devolvería la página `bajaView.html` de la carpeta `templates/usuarios`.

- Los métodos de los controladores pueden devolver otros objetos (además de String que representa a la vista) como pueden ser archivos JSON u objetos de tipo respuesta HTTP, todo ello lo veremos más adelante.

- En cuanto a la estructuración de los "mappings", una costumbre habitual, es definir a nivel clase controladora un `@RequestMapping` con la ruta base de ese controlador y luego `@GetMapping`, `@PostMapping`, etc. para cada uno de los métodos de la clase, sin necesidad de repetir la ruta base.

En el siguiente ejemplo, el `@RequestMapping` inicial hace que todas las rutas tratadas por el controlador se refieran a `"/app"`, por lo que el `@GetMapping "/misproductos"` manejaría la URL `"/app/misproductos"` y así con el resto de métodos del controlador:

```
package com.mycompany.myproject.controllers;
// imports . . .

@Controller
@RequestMapping("/app")
public class ProductosController {
    //Atributos. . .
    @GetMapping("/misproductos")
    public String list(Model model) {
        //proceso
        return "producto/listaView";
    }

    @GetMapping("/misproductos/{id}/eliminar")
    public String eliminar(@PathVariable Long id) {
        //proceso
        return "redirect:/app/misproductos";
    }

    @GetMapping("/producto/nuevo")
    public String nuevoProductoForm(Model model) {
        //proceso
        return "producto/formFormView";
    }

    @PostMapping("/producto/nuevo/submit")
    public String nuevoProductoSubmit(Producto producto) {
        //proceso
        return "redirect:/app/misproductos";
    }
}
```

Thymeleaf y contenido dinámico

La vista es la interfaz con la que interactuará el usuario, y su núcleo sea una plantilla HTML como acabamos de ver. También hemos comprobado en las páginas anteriores que todo el contenido es estático, no podemos parametrizarlo ni aportarle ninguna lógica de negocio. Para poder hacer esto, dotar de dinamismo nuestras vistas, necesitamos modificar nuestra aplicación en varios aspectos:

- Incluir en la vista ciertos “códigos” que no se muestren al usuario tal cual, si no que sean sustituidos por contenido dinámico en tiempo de ejecución.
- Pasar a la vista los valores o datos que sustituirán los códigos que acabamos de mencionar. Nos referiremos a este conjunto de datos como el *modelo de datos de la vista*.

Spring, dentro de todos sus proyectos, incluye uno llamado Spring MVC que se encarga de facilitarnos este aspecto. Spring MVC es independiente de la tecnología empleada para esta tarea, por defecto usamos Thymeleaf, un motor de plantillas poco invasivo en el HTML y de integración inmediata en SpringBoot, pero con pequeños cambios de configuración podríamos usar las antiguas JSP u otras tecnologías como FreeMarker, Groovy Markup, Apache Tiles, etc.

Thymeleaf va a tomar una plantilla HTML (podría ser también CSS, XML, etc.) junto con un modelo de datos (por ejemplo, datos tomados de una consulta de una base de datos) y generará una vista que será renderizada por el navegador. Por ejemplo, sobre la vista que contuviese:

```
<h1>Bienvenido
  <span th:text="${nombre}">nombre por defecto</span>
</h1>
```

Si le pasamos a esta vista el valor “*Pepe*” para la variable “*nombre*”, el navegador del usuario final recibiría:

```
<h1>Bienvenido
  <span>Pepe</span>
</h1>
```

“*Natural templating*” hace referencia a que no agrega nuevas etiquetas a la plantilla, sino que lo que aporta son nuevos atributos a etiquetas ya existentes. Así, si trabajamos con la plantilla sin pasar por el motor de plantillas, podemos renderizarlo de forma estática, sin procesar los atributos Thymeleaf y veríamos los textos por defecto. En el ejemplo anterior, mostraría:

```
<h1>Bienvenido
  <span>nombre por defecto</span>
</h1>
```

Así es muy cómodo trabajar con la plantilla, en la fase de diseño y maquetación, para luego añadir los atributos que dotarán de dinamismo a la plantilla.

No vamos a pararnos mucho en Thymeleaf ya que este curso se centra en la parte servidor y, en general, la parte cliente se desarrollará como aplicación independiente. Actualmente disponemos de infinidad de sistemas cliente (ordenadores, móviles, IoT, etc.) que pueden emplear software con distintas tecnologías: JavaScript y sus frameworks o bien otros lenguajes como Flutter, Kotlin, etc. por lo que no siempre la aplicación *backend* deberá servir una página HTML.

Nuestras aplicaciones *backend* deberían ser lo más flexibles posibles y encargarse solo de entregar los datos necesarios en un formato sencillo (por ejemplo JSON) para que sea la aplicación cliente, usando cualquier tecnología, quien se encargue de implementar la vista de usuario. Esta forma de trabajo la veremos en el capítulo de API Rest, pero por ahora trabajaremos con Thymeleaf, devolviendo HTML al cliente, para facilitar la visualización de los conceptos que vamos a ir viendo en cada capítulo.

Thymeleaf dispone distintos “dialectos”, es decir, dispone de distintas formas de añadir las expresiones a nuestra plantilla para ser renderizada. Aunque existe un dialecto por defecto, nosotros emplearemos uno propio de Spring: SpEL (Spring Expression Language).

Para emplear Thymeleaf solo debemos añadirle al proyecto la dependencia starter Thymeleaf, además de la dependencia web (al crearlo o luego en el pom.xml). Ello incluirá las librerías necesarias, pero también aplicará una configuración por defecto, por ejemplo, la ruta en la que estarán las plantillas y su extensión o sufijo por defecto.

```

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

```

Si no empleásemos SpringBoot deberíamos hacer toda esta configuración “a mano”, o bien mediante un archivo XML o bien mediante clases java:

```

@Configuration
public class ThymeleafConfiguration {

    @Bean
    public SpringResourceTemplateResolver templateResolver(){
        SpringResourceTemplateResolver templateResolver = new SpringResourceTemplateResolver();
        templateResolver.setApplicationContext(this.applicationContext);
        templateResolver.setPrefix("/WEB-INF/templates/");
        templateResolver.setSuffix(".html");
        templateResolver.setTemplateMode(TemplateMode.HTML);
        templateResolver.setCacheable(true);
        return templateResolver;
    }

    @Bean
    public SpringTemplateEngine templateEngine(){
        SpringTemplateEngine templateEngine = new SpringTemplateEngine();
        templateEngine.setTemplateResolver(templateResolver());
        templateEngine.setEnableSpringELCompiler(true);
        return templateEngine;
    }

    @Bean
    public ThymeleafViewResolver viewResolver(){
        ThymeleafViewResolver viewResolver = new ThymeleafViewResolver();
        viewResolver.setTemplateEngine(templateEngine());
        viewResolver.setOrder(1);
        viewResolver.setViewNames(new String[] {"html", "xhtml"});
        return viewResolver;
    }
}

```

Si queremos configurar ciertos aspectos del comportamiento de Thymeleaf podemos hacerlo mediante el archivo de configuración del proyecto: *application.properties* (es el archivo en el que previamente habíamos cambiado el puerto de escucha de la aplicación). Un parámetro típico a añadir es: **thymeleaf.cache**, por defecto está a *true* porque aumenta la velocidad, pero si la ponemos a *false*, cuando cambiemos solo la plantilla, no el controlador, podremos “refrescar” el navegador para ver los cambios sin necesidad de relanzar el proyecto (útil cuando estemos en desarrollo). Esto se denomina “hot swapping”.

`spring.thymeleaf.cache=false`

Ahora ya en nuestras plantillas tendremos que modificar la etiqueta `<html>` de nuestra vista para indicarle que será procesada por Thymeleaf. Para ello haríamos:

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

Etiquetas Thymeleaf básicas

Thymeleaf es una librería muy potente, con multitud de características que podemos incorporar en nuestras vistas, aunque como comentábamos previamente, este ámbito pertenece más al *Front-End* que a lo que nos interesa a nosotros, y cuando trabajemos con API Rest (que será lo habitual) no emplearemos Thymeleaf en absoluto.

Vamos a ver entonces, lo mínimo necesario para componer páginas MVC, pero con un tratamiento mínimo en el cliente, por ejemplo, aunque Thymeleaf dispone de funciones para pasar a mayúsculas un valor procedente del servidor, nosotros haremos el pase a mayúsculas en el servidor para que Thymeleaf solo lo muestre en la página.

La operación más básica es la que nos permite mostrar en una etiqueta un valor procedente del servidor. Para ello le añadiríamos a la etiqueta el atributo **th:text** la variable procedente del servidor entre \${...}

```
<h1 th:text="${city}">ciudad por defecto</h1>
```

En el caso de que el texto pasado sea una fecha (LocalDate, LocalDateTime) podemos formatearla:

```
<p th:text="#{temporals.format( fecha, 'dd-MM-yyyy HH:mm' )}">2024-12-31</p>
```

Condiciones

Otra operación básica consistiría en establecer una condición de forma que haya una parte de nuestro HTML que se muestre o no. Utilizaríamos **th:if** para la parte "if" y **th:unless** para el "else". Ejemplo:

```
<div th:if="${result>10}"> resultado mayor de 10 </div>
<div th:unless="${result>10}"> resultado no es mayor de 10 </div>
```

Y mostrando la variable *result*:

```
<div th:if="${result>10}"><span th:text="${result}">*</span> es mayor de 10</div>
<div th:unless="${result>10}"><span th:text="${result}">*</span> no es mayor de 10 </div>
```

Otra forma de implementar una condición es mediante el operador **?('elvis')**. Este operador se sitúa después de la condición y luego fijaríamos la sentencia en caso de que la condición se evalúe positivamente, luego dos puntos **:** y la sentencia en caso de que la condición se evalúe como negativa.

```
<span th:text="${result>10} ? 'Número mayor 10' : 'Número no mayor 10'">*</span>
```

Iteraciones

Para iterar sobre una colección, utilizaremos **th:each**. Contendrá una variable que será la que tome el valor que irá variando en cada iteración, a continuación, dos puntos y para terminar la colección sobre la que iterar entre \${...}. Luego la variable se tratará, por ejemplo, con **th:text**. Ejemplo:

```
<p th:each="producto:${listaProd}">
    <span th:text="${producto}">default product</span>
</p>
```

En este ejemplo *listaProd* podría ser un ArrayList de *String*, un array, un Set, etc.

Si en el caso anterior, el elemento pasado no es un tipo básico, como *String*, sino que es un objeto con atributos, podemos acceder a ellos siempre que la clase tenga getters con formato estándar: *getId()*, *getNombre()*, etc.

```
<table>
    <tr th:each="empleado:${listaEmpleados}">
        <td th:text="${empleado.id}">id</td>
        <td th:text="${empleado.nombre}">nombre</td>
        <td th:text="${empleado.email}">email@gmail.com</td>
        <td th:text="${empleado.salario}">0</td>
    </tr>
</table>
```

En algún caso puede interesarnos tener el número de iteración en la que estamos en cada momento, por ejemplo, para mostrarlo en la vista, a modo de numeración. Añadiríamos una variable a la iteración y tendríamos dos atributos que podemos usar sobre esa variable: **index**, que comienza en cero, o **count** que comienza en 1. Esta sería su sintaxis:

```
<tr th:each="empleado, iterador:${listaEmpleados}">
    <td th:text="${iterador.count}">contador</td>
    <td th:text="${empleado.nombre}">nombre</td>
</tr>
```

Fragmentos

Los fragmentos son bloques de código que podemos guardar en un archivo para poder reutilizar esos bloques en distintas páginas, evitando duplicidad de código. Un uso típico es el `<head>`, los pies de página, o los menús generales de la aplicación que se repetirán en varias páginas de nuestra web.

Cada fragmento se identifica mediante el atributo `th:fragment="nombre_fragmento"` pudiendo incluir varios fragmentos en un solo archivo. Si el fragmento va a contener código de la cabecera se incluirá en la parte `<head>` de ese archivo, y si el fragmento va a contener código del cuerpo, se incluirá en la parte `<body>`; es fácil de ver en el siguiente ejemplo.

```
<!DOCTYPE HTML>
<html xmlns:th="http://www.thymeleaf.org">
    <head th:fragment="cabecera">
        <meta charset="UTF-8" />
        <title>Título de todas las páginas</title>
        <link th:href="@{/css/styles.css}" rel="stylesheet">
    </head>
    <body>
        <footer th:fragment="pie">
            <a href="https://www.fernandowirtz.com">&copy;IES Fernando Wirtz</a>
        </footer>
    </body>
</html>
```

La ubicación por defecto de estos archivos será la carpeta `/templates` y si vamos a crear muchos archivos de fragmentos sería adecuado crear una subcarpeta a la que podemos llamar “fragments” o “common”. El nombre del fragmento puede ser el que deseemos, por ejemplo: `fragmentos.html`.

Una vez creados los fragmentos, en cada página de nuestra web que los queramos emplear, añadiremos `th:replace` para sustituir la etiqueta HTML indicada por el fragmento. A continuación, indicaremos la ruta del archivo + :: + nombre de fragmento. Por ejemplo:

```
<head th:replace="~{/fragmentos.html :: cabecera}"></head>
```

Que sustituiría esa etiqueta `<head>` por todo el fragmento tal y como lo construimos. En el caso de que no queramos sustituir la etiqueta original completamente por el fragmento, sino que queramos conservar lo que incluya y añadir el fragmento emplearemos `th:insert` en vez de `th:replace`. El problema de `th:insert` es que añade la etiqueta del fragmento, por lo que, en el caso del ejemplo anterior, tendríamos dos etiquetas `<head>`, la del documento original y la que inserta el fragmento. Para hacer inserts, emplearemos en el fragmento `th:block` en vez de la etiqueta HTML. Esta etiqueta nos sirve para agrupar el contenido del fragmento, pero no se traslada al documento final. Ejemplo:

```
<!DOCTYPE HTML>
<html xmlns:th="http://www.thymeleaf.org">
    <head>
        <th:block th:fragment="cabecera">
            <meta charset="UTF-8" />
            <title>Título de todas las páginas</title>
            <link th:href="@{/css/styles.css}" rel="stylesheet">
        </th:block>
    </head>
    <body>
        <footer th:fragment="pie">
            <a href="https://www.fernandowirtz.com">&copy;IES Fernando Wirtz</a>
        </footer>
    </body>
</html>
```

Y luego en la vista que queremos añadir el fragmento:

```
<head>
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <th:block th:insert="~{fragment.html :: cabecera}"></th:block>
</head>
```

Hay que señalar que `th:block` es la única etiqueta Thymeleaf que emplearemos durante el curso, con ese fin de agrupar líneas de código, el resto de elementos de Thymeleaf son atributos `th:` sobre etiquetas HTML.

Otro ejemplo de `th:block` -> imaginemos que queremos hacer una estructura repetitiva `th:each` sobre una colección y queremos que, para cada iteración, generar dos filas de una tabla. No podemos hacer `<tr th:each="elem:${colección}">` ya que solo generaría una fila en cada iteración. Haríamos:

```
<th:block th:each="elem : ${colección}">
  <tr><td th:text="${elem}"></td></tr>
  <tr><td th:text="${elem}"></td></tr>
</th:block>
```

Los fragmentos pueden tener incluso otras etiquetas Thymeleaf en su interior. Por ejemplo, si en el fragmento `"cabecera"` quisiésemos que el título fuese distinto para cada página podríamos incluir:

```
<title th:text="${titulo}"></title>
```

Obviamente, en el `Model` que a la vista que incluye el fragmento, debemos añadirle un atributo para esa variable, por ejemplo: `model.addAttribute("titulo", "Página principal");`

Tratamiento de nulos

Si el elemento que recibimos del servidor para una etiqueta contiene un valor nulo se producirá una excepción. Para evitar esto, Thymeleaf tiene varios mecanismos. El primero es mediante el operador `?` a continuación del objeto del que queremos verificar la existencia de nulos. En caso de ser nulo, mostrará la etiqueta vacía. Ejemplo:

```
<td th:text="${empleado?.nombre}">nombre</td>
```

En caso de que necesitemos que la etiqueta tome un valor en caso de nulos (no etiqueta vacía), podemos usar el operador `?.` que es similar al operador condicional `elvis` `?"` pero añadiendo solo la parte `else` que incluirá el contenido a mostrar en caso de nulos. Ejemplo:

```
<td th:text="${empleado.nombre}?:'sin nombre'">*</td>
```

Además de lo visto en este apartado, en los formularios tendremos características Thymeleaf específicas, las veremos en el capítulo 5.

Thymeleaf y CSS

Thymeleaf ofrece la posibilidad de incorporar o modificar dinámicamente atributos y clases CSS. Para ello disponemos de etiquetas como `th:style` (añadir a la etiqueta HTML una propiedad CSS) o `th:classappend` (añadir a la etiqueta HTML una clase CSS). En muchos casos incorporaremos dentro de una estructura condicional. Ejemplos:

```
<table th:style="${numero>0 ? 'display:block': 'display:none'}" > . . . </table>
```

En el ejemplo anterior, si el número pasado a la plantilla es mayor 0 añadirá el estilo `display='block'`, esto es, se mostrará la tabla, en caso contrario, se añadirá el estilo `display='none'`, esto es, la tabla no se mostrará.

```
<span th:classappend="${numero>50?'highNumber':'lowNumber'" th:text="${numero}">*</span>
```

En este caso, si el número pasado a la plantilla es mayor que 50 añadirá la clase CSS creada por nosotros `'highNumber'` a la etiqueta, y en caso contrario añadirá la clase `'lowNumber'`.

Paso de datos a la plantilla

En el apartado anterior hemos visto como situar en la plantilla etiquetas (más bien atributos sobre etiquetas HTML) para mostrar el contenido procedente del servidor. Ahora nos hace falta la parte complementaria de este tratamiento: cómo pasarle esos valores a la plantilla.

Para ello, al método del controlador le pasaremos como parámetro una instancia de una clase *Model* y antes de hacer el *return* de la vista, debemos añadirle a ese *model* los elementos que queremos pasarle a la vista.

Un Model (o ModelMap) es similar a un Map. Recordamos del curso anterior que un Map es una colección que almacena pares clave-valor, no posiciones numéricas como un arraylist. En este caso le pasaremos los pares: etiqueta thymeleaf + valor por el que sustituir la etiqueta. Para añadir esos pares al *Model* utilizaremos su método **addAttribute**.

Por ejemplo, para el primer ejemplo que vimos:

```
<h1 th:text="${city}">ciudad por defecto</h1>
```

Podríamos añadir este código en el controlador:

```
@GetMapping({ "/", "/home" })
public String showHome(Model model) {
    String ciudad = "A Coruña";
    model.addAttribute("city", ciudad);
    return "homeView";
}
```

En el ejemplo de iteración con Thymeleaf del apartado anterior:

```
<li th:each="producto:${listaProd}">
    <span th:text="${producto}">default product</span>
</li>
```

Esta podría ser una posibilidad para pasale datos a a vista.

```
@GetMapping("/productos")
public String showProducts(Model model) {
    List<String> listaProductos =
        new ArrayList<>(Arrays.asList("Pr1", "Pr2", "Pr3"));
    model.addAttribute("listaProd", listaProductos);
    return "productsView";
}
```

Rescatar parámetros de las URL

Una de las formas más habituales del paso de datos desde el cliente al servidor será mediante parámetros en la URL. Primero veamos las partes de una URL en el siguiente esquema.



Tendremos dos formas de pasar los parámetros en la URL:

- Pasar los parámetros en la parte de la *query*: después del *path*, después de la interrogación y separados por & en pares variable=valor.
Ejemplo: www.fernandowirtz.com?profesor=rdf&modulo=programación
- Pasar los parámetros en el “path” de la URL (el path es lo que va antes de “?”: normalmente incluye el protocolo, dominio y rutas separadas por /). Ejemplo: www.fernandowirtz.com/profesores/rdf

Para rescatar los valores proporcionados en la parte query usaremos `@RequestParam` y para los proporcionados en el path usaremos `@PathVariable`. Los vemos uno a uno.

@RequestParam

Esta anotación será un parámetro que le pasaremos al método del controlador y que incluirá el nombre de la variable en la URL y a continuación la variable que recogerá el dato de ese parámetro de la URL, y finalmente recordemos que también le pasábamos a estos métodos un *Model* para pasarle datos a la vista.

Con todo esto, un ejemplo típico de firma de controlador, por ejemplo, para la URL: www.fernandowirtz.com?profesor=rdf sería:

```
@GetMapping("/")
public String home (@RequestParam("profesor") String nombreProfesor, Model model){
```

Vemos como `@RequestParam` hace referencia al nombre de la variable. Ahora el String `nombreProfesor` contendría "rdf". Este tipo de datos suele ser String, pero podría ser de otro tipo: Long, Double, Date, etc.

Si tenemos dos parámetros en la query de la URL, repetiríamos la anotación, por ejemplo, para la URL: [www.fernandowirtz.com ?profesor=rdf & modulo=programación](http://www.fernandowirtz.com?profesor=rdf&modulo=programación) sería:

```
@GetMapping("/")
public String home (@RequestParam("profesor") String nombreProfesor,
                    @RequestParam("modulo") String nombreModulo,
                    Model model){
```

En el caso de que la variable que recoge el valor se llame como la variable en la URL simplificamos la notación, quedando así:

```
@GetMapping("/")
public String home (@RequestParam String profesor,
                    @RequestParam String modulo,
                    Model model){
```

Esas variables ahora las podemos usar en el cuerpo del controlador y pasárselas a la vista a través del model. Ejemplo:

```
@GetMapping("/")
public String home (@RequestParam String profesor,
                    @RequestParam String modulo,
                    Model model){
    model.addAttribute ("profe", profesor);
    model.addAttribute ("modul", modulo);
    return "index";
```

y ahora deberíamos añadir las expresiones Thymeleaf para mostrar ambos valores.

```
<!DOCTYPE html>
<html xmlns:th="http://www.thymeleaf.org">
<head><meta charset="UTF-8"></head>
<body>
    <p>El profesor <span th:text="${profe}">profesor por defecto</span>
    imparte el módulo de <span th:text="${modul}">módulo por defecto</span>
</p>
</body>
</html>
```

Un problema que nos encontraremos es que si el usuario invoca a la URL mapeada (en el ejemplo era la ruta raíz "/") sin parámetros, o no se ajustan a los parámetros que esperamos, se producirá un error 400 (URL mal formada). Para evitarlo podemos incluir a `@RequestParam` el parámetro `required=false` para que no se produzca el error. Adicionalmente podemos añadirle `defaultValue` para que si no introducen el parámetro la variable tome ese valor y no `null`.

Ejemplo:

```
@GetMapping("/")
public String home (
    @RequestParam (name="profesor", required=false)      String nombreProfesor,
    @RequestParam (name="modulo", required=false, defaultValue="X") String nombreModulo,
    Model model){
    if (nombreProfesor==null) nombreProfesor ="X";
    model.addAttribute ("profe", nombreProfesor);
    model.addAttribute ("modul", nombreModulo);
    return "index";}
```

o, más sencillo, con los nombres de variable iguales en URL y controlador:

```
@GetMapping("/")
public String home (
    @RequestParam (required=false) String profesor,
    @RequestParam (required=false, defaultValue="X") String modulo,
    Model model){
    if (profesor==null) profesor="X";
    model.addAttribute ("profe", profesor);
    model.addAttribute ("modul", modulo);
    return "index";
```

Hay otra forma de hacerlo, empleando la clase genérica `Optional` que recordemos del curso pasado que es una clase genérica, a forma de envoltorio de otra clase, en este caso `String`, que evita errores de ejecución por valores nulos y nos ofrece métodos para tratar esos nulos. En nuestro caso el método `orElse()` que devuelve el valor que le pasemos en caso de que el objeto sea nulo. Quedaría así:

```
@GetMapping("/")
public String home (
    @RequestParam Optional <String> profesor,
    @RequestParam Optional <String> modulo,
    Model model){
    model.addAttribute ("profe", profesor.orElse("X"));
    model.addAttribute ("modul", modulo.orElse("X"));
    return "index";
}
```

El método `get()` de `Optional` nos ofrecerá el valor y también puede ser útil el método `isEmpty()` para comprobar si el contenido tiene algún valor o es nulo:

```
if (profesor.isEmpty()) { . . . }
```

@PathVariable

Veremos ahora como rescatar valores de la URL cuando nos los pasan de la otra forma que comentábamos, en la parte *path* de la URL. En los últimos años está muy de moda ya que la URL queda más legible. Nos encontraremos entonces los valores separados por barras, y no necesariamente tienen que estar al final de la URL:

```
www.fernandowirtz.com/buscaprofe/rdf
www.fernandowirtz.com/buscaprofe/rdf/horario
```

Para tratarlo, incluimos en la anotación del controlador, junto con la ruta, una variable entre llaves que represente el valor que pretendemos capturar.

```
@GetMapping ("/buscaprofe/{nombreProfesor}")
```

Y en los parámetros del método, el parámetro `@PathVariable` con el mismo nombre.

```
@GetMapping ("/buscaprofe/{nombreProfesor}")
public String showProfe (@PathVariable String nombreProfesor, Model model){
    model.addAttribute ("profe", nombreProfesor.toUpperCase());
    return "profeView";
```

y deberíamos hacer la vista `profeView.htm` con la expresión Thymeleaf `#{profe}`.

En este caso no hay valores por defecto ni posible prevención de errores, porque si la URL no es correcta el servidor devolverá directamente un error 404.

Tanto en `@RequestParam` como `@PathVariable`, si definimos las variables como no `String`, por ejemplo, `Double`, `Integer`... y los valores introducidos por el usuario no son coherentes con ese tipo, se producirá un error. En algunos casos puede ser útil definirla como `String` y luego, dentro de un `try...catch` hacer la conversión al formato adecuado.

Ejemplo:

```
public String met(@PathVariable String num, Model model) {
    int numero;
    try {
        numero = Integer.parseInt(num);
    } catch (Exception e) {
        //redireccionar a página de error
    }
}
```

Retorno de los métodos del controlador

Por ahora, hemos visto que **un método de un controlador devuelve un String, que se corresponde con la vista que va a devolver al cliente** (sin la extensión *.html*).

Otra opción distinta sería la de redirigir hacia otro controlador, esto es muy frecuente por ejemplo si se produce un error, para redirigirlo al controlador que hará el tratamiento de errores, o también después de insertar un elemento en una base de datos, que redirija a un listado de elementos de esa base de datos, para verlo actualizado.

Para hacer esto, en el return ponemos: **redirect:** y la ruta de del controlador, tal y como estamos haciendo en las anotaciones @GetMapping.

```
@GetMapping("/par/{num}")
public String showPar (@PathVariable Integer num, Model model) {
    if (num < 1) return "redirect:/menuprincipal";
    model.addAttribute ("par",num%2);    //boolean
    return "parView";
}

@GetMapping("/menuprincipal")
public String showMenu (Model model) {
    model.addAttribute . . .      //variables para La página menuView
    return "menuView";
}
```

Sobre el ejemplo anterior, podría surgirnos la duda qué diferencia hay entre hacer: *return "redirect:/menuprincipal"* o directamente *"return menuView"*, si al final y al cabo es esa la vista a la que irá. La respuesta es que tal y como está el ejemplo, llegará a *menuView.html* a través del controlador y por tanto con las variables del *model*, en caso de hacerlo sin *redirect* llegaría a esa vista, pero sin *model*.

Además de una vista o una redirección a otro controlador, el controlador podrá devolver otros tipos de objetos como veremos en el capítulo de API Rest: (ResponseBody + cualquier tipo de datos), ResponseEntity, ResponseEntity, etc.

Antiguamente también se utilizaban objetos de tipo **ModelAndView** para devolver la vista y el modelo en un solo objeto, pero actualmente ha sido sustituido por el String que contiene la vista. En este caso se devolvía un *Model* pero uno de los atributos de ese *model* contenía la vista a mostrar.

```
@GetMapping("/user")
public ModelAndView showUser() {
    ModelAndView modelAndview = new ModelAndView();
    modelAndview.addObject("usuario", "Pepe");
    modelAndview.setViewName("userDetailView");
    return modelAndview;
```

Construcción de URLs dinámicas en la vista

En los enlaces de nuestras vistas tenemos las rutas que luego serán gestionadas por los controladores. Algunas de esas rutas son fijas en nuestra aplicación como la página de inicio, la página de "quienes somos", etc. pero habrá otras que será necesario construir dinámicamente, por ejemplo, imaginemos que la vista muestra una lista de productos obtenida de la base de datos y queremos al clicar en ellos visite la página del cada producto.

La expresión en Thymeleaf para referirse a URL es `@{...}` y entre las llaves tenemos distintas opciones dependiendo del tipo de enlace que necesitemos:

- **URL absolutas:** hacia otros servidores. Son rutas completas incluyendo el protocolo. En caso de no contener ninguna parte dinámica, en realidad, no es necesario Thymeleaf para ellas.
`<a th:href="@{http://www.thymeleaf/documentation.html}">`
- **URL relativas al contexto.** Son las más utilizadas, empiezan por `"/"` y esa barra representa la raíz de nuestra aplicación. Siguen por tanto una formación similar a los mappings que estamos viendo hasta ahora.
`<a th:href="@{/order/list}">`

Cuando el proyecto se empaqueta como `'jar'`, con el propio servidor Tomcat embebido, la aplicación se sitúa en la raíz del mismo, por lo que no hay diferencia entre `<a th:href="@{/order/list}">` y ``. Por el contrario, en empaquetado `'war'`, la aplicación puede situarse en otra ruta, en una subcarpeta, por ejemplo `"/app"`, por lo que con la primera opción obtendríamos una ruta: `/app/order/list`, distinta de `/order/list` que obtenemos en la segunda opción, sin Thymeleaf.

- Existen otras similares a las anteriores llamadas **"relativas al servidor"**, que como su nombre indica, son relativas al servidor global, en vez de al contexto. Empiezan por `~`:
`<a th:href="@{~/billing-app/showDetails.htm}">`
- Por último, existen otras llamadas **"relativas al protocolo"** para enlazar con recursos externos como hojas de estilos o scripts.
`<script th:src="@{//scriptserver.example.net/myscript.js}">...</script>`

Las que más emplearemos serán las segundas, las relativas al contexto. Podemos usar estas expresiones no solo con `th:href`, sino que podemos usarlas en cualquier lugar:

```
<form th:action="@{/order/processOrder}">
```

Añadir parámetros a URLs en la vista

Podemos añadir parámetros en la parte query de una URL con `@{...}` poniendo los parámetros entre paréntesis:

```
<a th:href="@{/product(id=3)}">
```

Que quedaría una vez procesado así:

```
<a href="/product?id=3">
```

Si son varios parámetros los sepáramos por comas:

```
<a th:href="@{/product(id=3, size='big')}">
```

Quedaría en el navegador:

```
<a href="/product?id=3&size=big">
```

Para añadir parámetros, pero en el path, seguimos el mismo proceso, pero añadimos la variable también en la posición del path, entre llaves:

```
<a th:href="@{/product/{id}/{size}(id=3,size='big')}">
```

Quedaría en el navegador:

```
<a href="/product/3/big">
```

Expresiones en las URLs en la vista

La potencia de las URL dinámicas viene dada porque, además de lo que acabamos de ver, podemos incluir variables dinámicas, tomadas del modelo que recibe la vista, en tiempo de ejecución.

Supongamos la siguiente URL:

```
<a th:href="@{/product/(id=3,size='big')}">
```

Pero que los valores que toma *id* (en este caso 3) y *size* (en este caso "big") no los conocemos previamente, se obtienen en tiempo de ejecución. Pues bien, podríamos sustituirlos por variables u otras expresiones:

```
<a th:href="@{/product(id=${myId} , size=${mySize})}"> <!--query-->
<a th:href="@{/product(id=${myId} , size=${mySize}>100) ?'big': 'small'}"> <!--query-->
<a th:href="@{/product/{id}/{size}(id=${myId} , size=${mySize})}"> <!--path-->
```

Desde el controlador enviaremos los valores para esas variables con instrucciones como:

```
model.addAttribute("myId", 3);
model.addAttribute("mySize", "big");
```

En todos estos ejemplos solo hemos puesto la parte de hipervínculo de la etiqueta pero hace falta añadir el texto sobre el que pulsar, y éste puede ser estático, no un simple texto, pero también puede ser dinámico.

Por ejemplo:

```
<a th:href="@{/product(id=${myId})}">Enlace al producto</a>
```

Pero también:

```
<a th:href="@{/product(id=${myId})}">Enlace a prod:<span th:text="${myId}">*</span></a>
```

Realmente lo que enviamos en el parámetro *th:href* es una cadena, y en Thymeleaf podemos hacer operaciones con cadenas, siendo la concatenación, con el operando '+', la más habitual. Esta concatenación puede ser de variables Thymeleaf como de literales (que reflejaremos entre comillas simples). Así pues, el ejemplo visto previamente:

```
<a th:href="@{/product(id=${myId} , size=${mySize})}">enlace</a>
```

Que si le pasábamos desde el controlador *myId=3* y *mySize='big'*, acababa generando en el navegador:

```
/product?id=3&size=big
```

Podríamos haberlos escrito así:

```
<a th:href="@{'/product?id='+$myId+'&size='+$mySize}">enlace</a>
```

Y lo mismo para el caso de variable en el *path*. El ejemplo:

```
<a th:href="@{/product/{id}/{size}(id=${myId},size=${mySize})}">enlace</a>
```

Podríamos escribirlo así:

```
<a th:href="@{'/product/'+$myId+'/'+$mySize}">enlace</a>
```

URL dinámicas con JavaScript

Una última opción para enviar parámetros a la URL es el empleo de JavaScript, esto será especialmente útil si los parámetros están en otros elementos, como por ejemplo cajas de texto, en vez de ser variables enviadas desde el controlador.

El siguiente ejemplo crea una URL similar a las de ejemplos anteriores (*/product/3/big*) pero tomando los valores de sendas cajas de texto.

```
<body>
  Id:<input id="ident" type="text">
  Size:<input id="size" type="text">
  <button onclick="enviarEnlace()">Enlace</button>
  <script>
    function enviarEnlace() {
      var txt1 = document.querySelector("#ident").value;
      var txt2 = document.querySelector("#size").value;
      window.location.href = "/product/" + txt1 + "/" + txt2;
    }
  </script>
</body>
```

Podríamos haber utilizado también `getElementById` para obtener el valor de las cajas de texto y `addEventListener` en vez de `onclick`.

La función JavaScript `window.location.href` envía al servidor la URL asignada. En este caso, la componemos por concatenación de cadenas que pueden ser tanto literales como variables. Sería un comportamiento similar a los formularios que veremos en un capítulo posterior. La diferencia entre este envío (como el de una etiqueta `<a>`) con respecto a los formularios, es el método HTTP empleado: en este caso es GET, mientras que el envío de formularios se hará generalmente mediante POST.

Resumen de Controlador + Vista

Como resumen, vamos una aplicación muy sencilla con un controlador y una vista para ver cómo se relacionan ambos. En concreto presenta una tabla con números generados al azar (initialmente vacía) y nos permite añadir nuevos números pulsando el enlace '*nuevo número*' o eliminar los generados previamente, pulsando el enlace '*delete*' que hay al lado de cada número.

El controlador `NumerosController.java` estará ubicado en la misma carpeta/paquete que la clase principal con el main (aunque podríamos crear una carpeta `Controllers`) y la vista `/listView.html` estará en la carpeta `/templates`.

Número	Operación
7	delete
51	delete
64	delete

Total números: 3

[Nuevo número](#)

```

src > main > java > com > example > numleatorios > J NumerosController.java > ...
1 package com.example.numleatorios;
2 import java.util.LinkedHashSet;
3 import java.util.Random;
4 import java.util.Set;
5 import org.springframework.stereotype.Controller;
6 import org.springframework.ui.Model;
7 import org.springframework.web.bind.annotation.GetMapping;
8 import org.springframework.web.bind.annotation.PathVariable;
9
10 @Controller
11 public class NumerosController {
12     Random random = new Random();
13     public Set<Integer> lista = new LinkedHashSet<>();
14
15     @GetMapping({"/", "/list", ""})
16     public String showList(Model model) {
17         model.addAttribute("cantidadTotal", lista.size());
18         model.addAttribute("listaNumeros", lista);
19         return "listView";
20     }
21
22     @GetMapping("/new")
23     public String showNew() {
24         boolean añadido;
25         do { añadido = lista.add(random.nextInt(100) + 1);
26         } while (!añadido);
27         return "redirect:/list";
28     }
29
30     @GetMapping("/delete/{id}")
31     public String showDelete(@PathVariable Integer id) {
32         lista.remove(id);
33         return "redirect:/list";
34     }
35 }

```

```

src > main > resources > templates > listView.html > listView.html
1 <!DOCTYPE html>
2 <html lang="es" xmlns:th="http://www.thymeleaf.org">
3     <head></head>
4     <body>
5         <h1>Listado de números aleatorios</h1>
6         <table border="1px">
7             <thead><th>Número</th><th>Operación</th></thead>
8             <tbody>
9                 <tr th:each="numero : ${listaNumeros}">
10                     <td th:text="${numero}"></td>
11                     <td>
12                         <a th:href="@{/delete/{id}(id=${numero})}">delete</a>
13                     </td>
14                 </tr>
15             </tbody>
16         </table>
17         Total números: <span th:text="${cantidadTotal}">0</span><br/>
18         <a th:href="@{/new}">Nuevo número</a><br/>
19     </body>
20 </html>

```

Controlador (líneas 15-20): Este método del controlador se ejecutará como respuesta a peticiones como <http://localhost/>, <http://localhost/list> o <http://localhost>. Añadirá al modelo dos valores para enviárselos a la vista: *cantidadTotal* que contendrá la cantidad de elementos del LinkedHashSet y *listaNumeros* que será el contenido completo del LinkedHashSet. Finalmente invoca a la vista */templates/listView.html*

Vista: listView.html: en la línea 2 indicamos que tiene que procesar los atributos Thymeleaf. En la línea 18 escribe el total de líneas (el total de elementos del HashSet en el servidor), pero antes compone una tabla dinámicamente con una fila *<tr>* por cada número en *listaNumeros*, que es el mismo LinkedHashSet. En la primera celda de cada fila pone el número y en la segunda pone una URL: */delete/X* siendo X el número del Set que estamos tratando en ese momento. Finalmente, en la línea 16 tiene un enlace a */new* (recordemos que es alGetMapping */new* del controlador, no a una vista)

Controlador (líneas 22-28): Este método del controlador se ejecutará como respuesta a peticiones como <http://localhost/new> y que vendrá del enlace de la vista 'nuevo número' añadirá un nuevo número aleatorio al Set y nos redirige al GetMapping */list*, esto es, el que llamará a la vista inicial que muestra la tabla de números.

Controlador (líneas 30-34): Este método del controlador se ejecutará como respuesta a peticiones como <http://localhost/delete/X> que vendrán de los enlaces de la segunda columna de la tabla. Lo primero que hace es tomar el número X (a través de la anotación *@PathVariable*) y lo elimina del HashSet. Luego, al igual que el método anterior, nos redirige al GetMapping */list*

Nota: el nombre de los métodos del controlador no es relevante, por defecto podemos hacer que empiecen por 'show' o por 'get'. En cuanto a las vistas, quizás el que terminen en "View" puede ayudarnos a diferenciarlas bien de las partes de la URL que usamos en los GetMapping de los controladores.

Interfaz WebMvcConfigurer

Esta interfaz permite configurar aspectos básicos del funcionamiento de nuestra aplicación. Más adelante la emplearemos para evitar problemas de seguridad CORS, pero también podemos indicarle mappings directos entre una ruta y la vista que se mostrará, evitando tener que desarrollar el método controlador. Solo será válido cuando no se le pase a la vista ningún dato, es decir cuando sea una vinculación directa entre ruta y vista, sin nada más.

Para implementar esta interfaz, crearemos una clase con la anotación `@Configuration` (para que se ejecute al iniciar la aplicación) y que implemente los métodos de configuración que nos interesen, en este caso: `addViewController()`:

```
@Configuration
public class WebMvcConfig implements WebMvcConfigurer {
    @Override
    public void addViewControllers(ViewControllerRegistry registry) {
        registry.addViewController("/myURL").setViewName("myView");
    }
}
```

La clase puede tener el nombre que quieras. En este caso, si el usuario solicita la ruta `/myURL`, le mostrará la vista `/myView.html`.

Esta clase, como todas las de nuestro proyecto, debe estar en el paquete raíz o subpaquete. En algunos casos verás que esta clase también está anotada con `@EnableWebMvc`, pero, con SpringBoot no es necesario.

Páginas de error personalizadas

Como ya habrás comprobado, si se produce un error en nuestra aplicación, el navegador nos muestra una página de error por defecto. Los errores típicos suelen ser 404 (página no encontrada), 500 (Error interno del servidor), 403 (acceso denegado), etc.

Si queremos que muestre páginas personalizadas con un mensaje más amigable, con enlaces de vuelta a la aplicación, etc. basta con crear una carpeta llamada `/error` debajo de `/templates` y dentro de ella crear archivos `.html` con nombre igual al número de error, esto es, `404.html`, `500.html`, `403.html`

Tema 4: Servicios

Definición de Servicio

Como ya comentamos al principio del manual, la capa de servicio se encarga de la lógica de negocio, de las reglas y cálculos que la aplicación debe hacer en cada caso.

Típicamente los métodos de una clase de servicio son invocados desde un controlador y es éste último el que le pasa como parámetro al servicio los valores proporcionados por el cliente para hacer los cálculos necesarios. Es también común que el servicio necesite acceder a repositorios de datos, para obtener información adicional para ejecutar esas reglas de negocio (pero eso lo veremos en capítulos posteriores). Finalmente, los métodos de la capa de servicio devolverán el resultado al controlador para que lo envíe al cliente.

Las clases de servicio se anotan con `@Service` y en vez de crear instancias en el controlador con un constructor como haríamos con otras clases, Spring **inyecta el servicio en el controlador mediante la anotación `@Autowired`**, evitando así el **acoplamiento de código**.

Acoplamiento de código e inyección:

Supongamos que tenemos una clase 'Coche' que tiene como atributo una clase llamada 'Motor'. Cuando instanciamos un coche tendremos que instanciar también su motor mediante uno de los constructores de Motor. En esta situación decimos que 'Coche' y 'Motor' están **fuertemente acoplados**: cualquier cambio en el constructor de 'Motor' haría que el constructor de 'Coche' fallase. El problema se agrava si pensamos que un coche tiene multitud de componentes además del motor: ruedas, volante, etc.

Una forma de solucionar este problema es "inyectar" la instancia de Motor en el constructor de Coche pasándoselo como parámetro, así evitamos los problemas en Coche si hay cambios en el constructor de Motor.

CODIGO ACOPLADO:

```
class Coche {
    String modelo;
    Motor motor;
    Coche (String mod, int idm, int pot) {
        this.modelo = mod;
        this.motor = new Motor (idm,pot);    }
}

class Motor {
    int id
    int potencia;

    Motor (int i, int p)  {
        this.id=i; this.potencia=p;    }
}
```

CODIGO DESACOPLADO:

```
class Coche {
    String modelo;
    Motor motor;
    Coche (String mod, Motor mot) {
        this.modelo = mod;
        this.motor = mot;    }

class Motor {
    int id
    int potencia;

    Motor (int i, int p)  {
        this.id=i; this.potencia=p;    }
}
```

Es una entidad externa (en nuestro caso el propio framework) el que hace la instancia de Motor para pasárselo al constructor de Coche.

En cuanto a la ubicación de los servicios, podemos crear un paquete (carpeta) llamada *services* y meterlos todos ahí, o si hemos optado por una organización más orientada a dominio, irán en la misma carpeta que los controladores de la misma área (*Cliente*, *Producto*, etc.) Spring no pone limitaciones en cuanto a esa organización, pero deben ser sub-paquetes del paquete raíz, como ocurría con los controladores.

Este sería un ejemplo de estructura de un controlador + servicio que recibe en la URL dos números y devuelve una vista con su suma.

En el servicio simplemente creamos el método con la operación solicitada:

```
package com.misproyectos.proyecto1.services;
import org.springframework.stereotype.Service;
```

```
@Service
public class SumaService {
    public Integer suma(Integer a, Integer b) {
        return a + b;
    }
}
```

En el controlador inyectamos el servicio e invocamos a su método.

```
package com.misproyectos.proyecto1.controllers;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Controller;
import org.springframework.ui.Model;
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.PathVariable;
import org.springframework.web.bind.annotation.RequestParam;

import com.misproyectos.proyecto1.services;

@Controller
public class SumaController {

    @Autowired
    private SumaService sumaService;

    @GetMapping("/suma/{numX}/{numY}")
    public String showSuma(
        @PathVariable Integer numX, @PathVariable Integer numY, Model model) {

        Integer result = sumaService.suma(numX, numY);
        model.addAttribute("resultado", result);
        return "resultSumaView";
    }
}
```

Podemos fijarnos en el ejemplo que:

- Ambos paquetes están debajo del mismo paquete raíz (*proyecto1*).
- Al añadir `@Autowired` no llamamos al constructor del servicio:

```
SumaService sumaService = new SumaService();
```

Más adelante veremos cómo podemos incluso evitar la anotación `@Autowired` usando [Lombok](#)

- Si hubiese que validar los datos de entrada, se podría hacer en otro método de este servicio o de otro servicio y ser invocado previamente desde el controlador.

Los objetivos principales de esta capa de servicio son:

- Separar la lógica de negocio de la vista de los datos. Cualquier cambio en la forma de recibir los datos y mostrarlos (por ejemplo, cuando veamos API Rest) no afectará a esta parte de la aplicación.
- Código más estructurado, métodos y clases más pequeños.
- Facilidad para testing unitario: podemos hacer pruebas con los diferentes servicios sin necesidad de simular todo el proceso de interacción con el usuario.

Gestión de errores

Los servicios pueden sufrir en distintas situaciones de error: puede ser porque los parámetros recibidos no tengan los valores esperados, o porque los cálculos que pretende realizar no se puedan llevar a cabo, o porque cuando accede a repositorios de datos, estos datos no estén disponibles, o bien muchas otras razones.

Cuando un servicio se encuentra en una situación de error, lo más correcto es que lance una excepción (`throw`) y que esta excepción sea capturada (`try...catch`) por el método que llamó al servicio, que generalmente será un método de controlador o bien otro método de servicio.

El siguiente ejemplo muestra una clase de servicio, con un solo método, que calcula el valor de la hipotenusa a partir de sus dos catetos. En el caso de que alguno de los catetos sea negativo lanza una excepción.

```
@Service
public class MathService {
    public Double calcularHipotenusa (Double cat1, Double cat2) throws RuntimeException {
        if (cat1 <= 0 || cat2 <= 0)
            throw new RuntimeException("Error en parámetros de entrada");
        return Math.hypot(cat1, cat2);
    }
}
```

Podríamos haber creado una excepción propia en vez de lanzar la genérica `RuntimeException` pero por ahora, el planteamiento visto es suficiente. Ahora, desde el controlador, se llamará al método de servicio capturando la excepción. En este ejemplo los datos de los catetos provienen de variables en el `path` de la URL:

```
@GetMapping("/calcularHipotenusa/{cat1}/{cat2}")
public String showHipot(@PathVariable Double cat1, @PathVariable Double cat2, Model model){
    try {
        model.addAttribute("resultado", mathService.calcularHipotenusa(cat1,cat2));
        return "indexView";
    } catch (RuntimeException ex) {
        model.addAttribute("txtError", ex.getMessage());
        return "errorView";
    }
}
```

En este caso, al capturar la excepción, envía al usuario una vista de error que informaría al usuario del problema ocurrido a través de esa variable `"txtError"`. No tiene por qué ser una vista propia para los errores, sino que en la vista de datos (en este caso `indexView.html`) podemos añadir esa misma variable y, ya en Thymeleaf, mostrarla en caso de que no sea nula.

Este planteamiento no funciona si la respuesta devuelta por el controlador no es una vista, si no que hace un redirect a otro `mapping`, ya que en este caso no podemos pasar nada por el `model`. Una solución para esta situación sería pasarle al mapping mediante parámetro un código de error (o incluso el propio texto de la excepción) y que sea éste el que lo procese. En la excepción haríamos:

```
try {
    ...
} catch (RuntimeException ex) { return "redirect:/home?err=1";}
```

Y en el controlador que muestra la vista:

```
@GetMapping("/home")
public String showHome(@RequestParam (required=false) Integer err, Model model){
    if (err!=null) model.addAttribute("txtErr", "texto del error 1");
    return "indexView";
}
```

Un inconveniente de esta opción es que “perdemos” el mensaje de la excepción generada, ya que por culpa del *redirect*, al *mapping* solo le llegan los parámetros, pero no el *model*.

Una solución a este problema sería que el controlador tuviese una variable global accesible por todos los *mapping* y que el *try...catch* le asigne el mensaje de la excepción a esa variable: así el *mapping* que devuelve la vista puede incorporar al *model* el texto de la excepción a través de esa variable. Ejemplo:

```
@Controller
public class MyController {
    @Autowired
    private MyService myservice;

    private String txtError;

    @GetMapping("/operacion")
    public String showOperacion(){
        try {
            myService.operacionConExcepciones();
        } catch (RuntimeException ex) { txtError=ex.getMessage(); }
        return "redirect:/home";
    }
    @GetMapping("/home")
    public String showHome(Model model){
        if (txtError!=null) {
            model.addAttribute("txtErr", txtError);
            txtError=null;           //vacía la variable para poder usarla de nuevo
        }
    }
}
```

Vemos como esa variable puede servir no solo para mensajes de error, sino para cualquier tipo de texto variable que queramos incorporar (podríamos llamarla *status* o similar).

Vamos a hacer el ejemplo completo del cálculo de la hipotenusa con este sistema, de forma que la aplicación informe mediante esta variable, bien del resultado, bien de la excepción producida.

```
@Controller
public class MathController {
    @Autowired
    private MathService mathService;

    private String txtStatus=null;

    @GetMapping("/")
    public String showInit(Model model) {
        if (txtStatus != null) {
            model.addAttribute("txtStatus",txtStatus);
            txtStatus=null;}
        return "indexView";
    }

    @GetMapping("/calcularHipotenusa/{cat1}/{cat2}")
    public String showHipot(@PathVariable String cat1,@PathVariable String cat2,
                           Model model) {
        Double cateto1, cateto2;
        try { cateto1 = Double.parseDouble(cat1);    //puede Lanzar excepción
               cateto2 = Double.parseDouble(cat2);
               model.addAttribute("resultado",
                                 mathService.calcularHipotenusa(cateto1, cateto2)); //otra excepción
               return "resultadoView";
        } catch (Exception ex) {txtStatus=ex.getMessage();}
        return "redirect:/";
    }
}
```

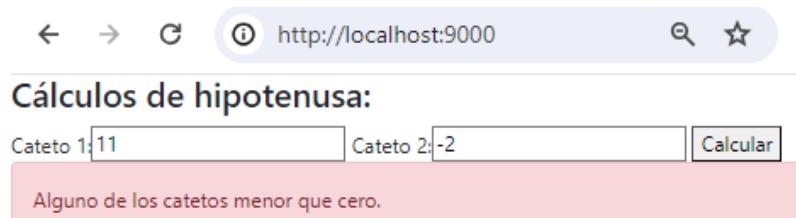
Para probarlo hacemos una vista *indexView.htm* con el siguiente código (recuerda añadir las *webjars* en el *pom.xml* para visualizarlo correctamente) en el que se introducen los dos catetos y una función JavaScript envía al servidor la petición para calcular la hipotenusa. Si se produce alguna excepción, se redirige de nuevo a esta vista, pero con el texto de la excepción.

```
<!DOCTYPE html>
<html xmlns:th="http://www.thymeleaf.org">
  <head>
    <meta charset="UTF-8">
    <meta http-equiv="X-UA-Compatible" content="IE=edge">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <link href="/webjars/bootstrap/css/bootstrap.min.css" rel="stylesheet">
    <script src="/webjars/bootstrap/js/bootstrap.bundle.min.js"></script>
    <title>Calcular hipotenusa</title>
  </head>
  <body>
    <h3>Cálculos de hipotenusa:</h3>
    Cateto 1:<input type="text" id="cateto1"/>
    Cateto 2:<input type="text" id="cateto2"/>
    <button onclick="calcularHipotenusa()">Calcular</button>
    <div th:if="${txtStatus!=null}">
      <p class="alert alert-danger" role="alert" th:text="${txtStatus}">error</p>
    </div>
    <script>
      function calcularHipotenusa() {
        var cateto1 = document.querySelector("#cateto1").value;
        var cateto2 = document.querySelector("#cateto2").value;
        window.location.href = "/calcularHipotenusa/" + cateto1 + "/" + cateto2;
      }
    </script>
  </body>
</html>
```

Y necesitamos una segunda vista llamada *resultadoView.htm* para mostrar el resultado:

```
<!DOCTYPE html>
<html xmlns:th="http://www.thymeleaf.org">
  <head>
    <meta charset="UTF-8">
    <meta http-equiv="X-UA-Compatible" content="IE=edge">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <link href="/webjars/bootstrap/css/bootstrap.min.css" rel="stylesheet">
    <script src="/webjars/bootstrap/js/bootstrap.bundle.min.js"></script>
    <title>Resultado hipotenusa</title>
  </head>
  <body>
    <p class="alert alert-primary" role="alert">
      Resultado:<span th:text="${resultado}">resultado</span>
    </p>
  </body>
</html>
```

En caso de error veríamos algo así:



Servicios mediante Interfaces y Clases

Si buscas ejemplos de servicios en internet verás que, en muchos casos, en vez de utilizar directamente una clase con la anotación `@Service`, tal y como hemos hecho con nuestra clase `SumaService`, lo que hacen es crear una interfaz con la firma de los métodos y una clase que implementa la interfaz.

La clase estaría anotada con `@Service`, pero en el controlador se inyecta la interfaz. Ejemplo:

```
public interface SumaService {
    Integer suma(Integer a, Integer b);
}
```

```
@Service
public class SumaServiceImpl implements SumaService {
    public Integer suma (Integer a, Integer b) {
        return a + b;
    }
}
```

```
@Controller
public class SumaController {
    @Autowired
    private SumaService sumaService;
    . . .
}
```

Puede parecer un poco confuso al principio y no verle la utilidad. La explicación es que de esta forma logramos más independencia entre las capas, como comentábamos en el primer capítulo de este manual. Si en un futuro quiero hacer una nueva implementación de la interfaz con otra clase, no tendré que cambiar nada en las otras capas como el controlador, ya que tiene inyectada la interfaz y no la clase.

```
@Service
public class SumaServicePositivos implements SumaService {
    public Integer suma (Integer a, Integer b) {
        if (a < 0 || b < 0 ) return -1;
        return a + b;
    }
}
```

En el capítulo siguiente veremos un ejemplo muy claro: haremos un servicio a gestión de Empleados (altas, bajas, modificaciones, etc.) sobre un `ArrayList` en memoria. Tendremos:

- Interfaz de servicio: `EmpleadoService` con métodos `añadirEmpleado`, `borrarEmpleado`, etc.
- Clase de servicio: `EmpleadoServiceImplMem` (implementa la interfaz anterior con `ArrayList`)
- Clase controlador: `EmpleadoController`, con `@Autowired EmpleadoService`

Luego, en el capítulo siguiente a ese, pasaremos esa gestión de hacerla en memoria con el `ArrayList` a hacerla mediante una base de datos, y veremos como no tendremos que hacer ningún cambio en el controlador, solo cambiaremos:

- Clase de servicio: `EmpleadoServiceImplBD` (implementa la interfaz anterior con base de datos) y no habrá que cambiar nada ni en la interfaz de servicio ni en la clase controlador.

Sobre este planteamiento surgen dos dudas:

Duda 1: ¿Es necesario este esquema de Interfaz y Clase?

La respuesta obviamente es que no, no es necesario, pero si lo hacemos, logramos:

- Código más estructurado, más independencia entre las capas de la aplicación.
- Facilidad para testing (es más fácil sustituir clases reales por "mocks" si empleamos interfaces).
- Separación de entornos: en una fase de desarrollo podemos trabajar con una clase y en producción sustituir por otra, siempre manteniendo la misma interfaz.
- Seguridad: la interfaz representa únicamente una declaración de intenciones. No aparece código por ninguna parte. Al exponer una interfaz no estás comprometiendo el código fuente del servicio.
- Como colofón a estas consideraciones está el **desacoplamiento de código**, del que seguiremos hablando a lo largo del manual.

Duda 2: Si lo que inyectamos en el Controlador es la interfaz ¿Qué ocurre cuando tenemos varias implementaciones de una interfaz, cual elige el controlador?

Lo habitual es tener una sola implementación. Si tenemos varias clases, podemos indicar que una de ellas es por defecto, aplicándole la anotación `@Primary`. Así la empleará por defecto. Cuando queramos aplicar otra clase podemos inyectarla directamente, aunque queda más elegante añadirles a las clases la anotación `@Qualifier` con un calificador e indicarle dicho calificador en la inyección.

```

@Service
@Primary
@Qualifier ("mem")
public class EmpleadoServiceImpl implements EmpleadoService { . . . }

@Service
@Qualifier ("bd")
public class EmpleadoServiceImplBD implements EmpleadoService { . . . }

@Controller
public class EmpleadoController {
    @Autowired
    @Qualifier ("mem")
    private EmpleadoService empleadoService;
    . . .
}

```

Nunca se puede anotar con `@Service` (o `@Component` o cualquier otra anotación derivada de `@Component`) una interfaz ya que cuando Spring trate de instanciarla se producirá un error.

CommandLineRunner

`CommandLineRunner` es una interfaz funcional con un solo método llamado `run`. Spring Boot llamará automáticamente al método de ejecución de todos los `beans` que implementen esta interfaz después de que se haya cargado el contexto de la aplicación, es decir en el arranque de nuestro programa.

En muchos casos se suele incluir en el archivo que contiene el `main` de la aplicación y la anotación `@SpringBootApplication`. Como interfaz funcional que es, se puede implementar de forma sencilla mediante una función lambda. Esta será la forma que empleemos habitualmente:

```

@SpringBootApplication
public class Main {
    public static void main(String[] args) {
        SpringApplication.run(Main.class, args);
    }
    @Bean
    public CommandLineRunner metodo(argumentos) {
        return args -> {
            //código
        };
    }
}

```

En los argumentos del método podemos inyectar componentes, pero lo veremos más adelante.

```
@Bean
public CommandLineRunner initData(ProductService ps) {
    return args -> {
        ps.añadir(new Product("Prod 1", 100.0f));
        ps.añadir(new Product("Prod 2", 200.0f));
    };
}
```

También puede ser la propia aplicación SpringBoot la que implemente la interfaz:

```
@SpringBootApplication
public class MyApp implements CommandLineRunner {
    public static void main(String[] args) {
        SpringApplication.run(MyApp.class, args);
    }
    @Override
    public void run(String... args) throws Exception {
        System.out.println("Welcome to our app!");
    }
}
```

O incluso en archivos aparte, como clases que implementan la interfaz:

```
@Component
public class MyRunner implements CommandLineRunner {
    @Override
    public void run(String... args) throws Exception {
        . . .
    });
}
```

Tema 5: Formularios

Desarrollo de formularios

Spring (junto con Thymeleaf) permite una gestión muy sencilla de los formularios. La forma de trabajar será asociando al formulario HTML un objeto llamado *CommandObject* que será un reflejo de los campos de dicho formulario, es decir, será **una clase que tendrá un atributo por cada campo**. Ese objeto puede ser una entidad de nuestro modelo, es decir corresponderse con un Producto, Cliente, etc. o ser una clase creada expresamente para esta función.

La gestión del formulario se hace en tres pasos:

- 1) Crear un controlador que invoca mediante `@GetMapping` a la plantilla que contiene el formulario, pasándole el objeto *CommandObject* que contendrá los campos del formulario.
- 2) En la plantilla HTML que tiene el formulario, le asociamos ese objeto que contendrá los campos mediante `th:object` y hacemos uno a uno la asociación: campo de formulario – atributo del objeto mediante el atributo Thymeleaf `th:field`. El submit de ese formulario nos enviará al `@PostMapping` del controlador indicado mediante `th:action`, donde se procesará el formulario.
- 3) En el `@PostMapping` del mismo controlador recogemos los datos del objeto y los procesamos y generalmente redirigimos al usuario hacia otro controlador (inicio, listado, etc...).

Recordemos que en el protocolo HTTP los formularios se envían generalmente mediante POST, no GET.

Vamos a verlo en un ejemplo:

Paso 1: Crear el commandObject y añadirlo al controlador en la presentación del formulario.

```
@GetMapping ("/myForm")
public String showForm (Model model) {
    model.addAttribute ("formInfo", new FormInfo());
    return "formView";                                //vista con el formulario
}
```

En este caso el *commandObject* no es una instancia de una clase de nuestro modelo, es de una clase a la que hemos llamado *FormInfo*, por lo que debemos crearla (puede ser en el mismo paquete que los controladores, o en otro cualquiera).

```
public class FormInfo {
    private String nombre;
    private Integer edad;
    //importante: Los atributos deben tener getters públicos con nombre estándar
    public String getNombre () { return nombre; }
    public Integer getEdad () { return edad; }
    public void setNombre (String nombre) { this.nombre = nombre; }
    public void setEdad (Integer edad) { this.edad = edad; }
}
```

Paso 2: Crear la plantilla: formView.html con el formulario y la referencia al objeto.

```
<form action="#" method="post"
    th:action="@{/myForm/submit}"      <!-- ruta del PostMapping -->
    th:object="${formInfo}">          <!-- nombre del objeto en el controller-->
    ...
    <input type="submit" value="Enviar"/>
</form>
```

En la segunda línea vemos el controlador al que se redirigirá cuando se envíe el formulario y en la tercera línea el nombre del objeto que contendrá los valores introducidos por el usuario.

Luego, para cada campo de formulario le indicamos el atributo del objeto *CommandObject* con el que se corresponde:

```
<input type="text" id="nombre" th:field="*{nombre}">
<input type="text" id="edad" th:field="*{edad}">
```

En este caso `*{nombre}` y `*{edad}` son atributos de la clase que sea `FormInfo`. Si la etiqueta HTML del elemento incluye el atributo `name` con el mismo nombre que el atributo en la clase del objeto `commandObject`, no sería obligatorio `th:field`.

Paso 3: Recoger los datos del formulario en el controlador, en la ruta indicada en el submit del formulario y con una anotación `@PostMapping`.

```
@PostMapping("/myForm/submit")
public String showMyformSubmit(FormInfo formInfo) {
    //tratamiento de los datos recibidos
    return . . .; //vista o redirect a donde queramos dirigir la respuesta
}
```

Si necesitamos pasar los datos recibidos a la vista a mostrar, podemos hacerlo de dos formas: como se vio en el tema 3: añadiendo el parámetro `Model` y la instrucción `model.addAttribute` o bien añadiendo la anotación `@ModelAttribute` al parámetro que representa el objeto del formulario; esta anotación inyecta directamente el objeto en el modelo de datos de la vista:

```
public String showMyformSubmit(FormInfo formInfo, Model model) {
    //tratamiento de los datos recibidos
    model.addAttribute("formInfo", formInfo);
    return "formSubmitView";
}

public String showMyformSubmit(FormInfo formInfo) {
    //tratamiento de los datos recibidos
    model.addAttribute("formInfo", formInfo); //si necesitamos estos datos en la vista
    return "formSubmitView";
}
```

```
MainController.java
1 import org.springframework.web.bind.annotation.PostMapping;
2
3 @Controller
4 public class MainController {
5     @GetMapping("/myForm")
6     public String showForm(Model model) {
7         model.addAttribute("formInfo", new FormInfo());
8         return "formView";
9     }
10
11     @PostMapping("/myForm/submit")
12     public String showMyformSubmit(FormInfo formInfo, Model model) {
13         formInfo.setNombre(formInfo.getNombre().toUpperCase());
14         model.addAttribute("formInfo", formInfo);
15         return "formSubmitView";
16     }
17 }
```

```
FormInfo.java
1 package com.proyectospring.myapp;
2
3 public class FormInfo {
4     private String nombre;
5     private Integer edad;
6
7     public String getNombre() {return nombre; }
8     public Integer getEdad() {return edad; }
9     public void setNombre(String nombre) { this.nombre = nombre; }
10    public void setEdad(Integer edad) { this.edad = edad; }
11 }
```

```
formView.html
1 <!DOCTYPE html>
2 <html xmlns:th="http://www.thymeleaf.org">
3     <head>
4         <meta charset="UTF-8">
5         <meta http-equiv="X-UA-Compatible" content="IE=edge">
6         <meta name="viewport" content="width=device-width, initial-scale=1.0">
7         <title>Test</title>
8     </head>
9     <body>
10        <form action="#" method="post" th:action="@{/myForm/submit}"
11             th:object="${formInfo}">
12            <label>Nombre: <input type="text" id="nombre" th:field="*{nombre}">
13            <label>Edad:<input type="text" id="edad" th:field="*{edad}"></label>
14            <input type="submit" value="Enviar"/>
15        </form>
16    </body>
17 </html>
```

```
formSubmitView.html
1 <!DOCTYPE html>
2 <html xmlns:th="http://www.thymeleaf.org">
3     <head>
4         <meta charset="UTF-8">
5         <meta http-equiv="X-UA-Compatible" content="IE=edge">
6         <meta name="viewport" content="width=device-width, initial-scale=1.0">
7         <title>Test</title>
8     </head>
9     <body>
10        <h1>Formulario procesado</h1>
11        <p>Nombre:<span th:text="${formInfo.nombre}"></span></p>
12        <p>Edad:<span th:text="${formInfo.edad}"></span></p>
13    </body>
14 </html>
```

Nota: Tal y como está en el ejemplo, la vista devolverá al servidor en el @PostMapping el objeto con nombre *formInfo*, y con este nombre lo trataremos en el método del controlador, pero si quisiésemos renombrarlo en la llegada, podríamos especificárselo en la firma del método:

```
@ModelAttribute("formInfo") FormInfo datosFormulario
```

De todas formas, lo más probable es que los datos recibidos del formulario sean tratados por la capa de servicio de nuestra aplicación para procesarlos de alguna forma. Finalmente mostrariamos una vista en la que informaríamos al usuario que el formulario ha sido procesado (o podríamos redirigirnos la salida a otro controlador).

Anotación @ModelAttribute:

Esta anotación se emplea en los controladores y tiene dos usos: anotando un parámetro de un método de un *request* del controlador o anotando todo un método.

- Si está en un parámetro de un método: inyecta ese parámetro directamente en el modelo de la vista que devolverá el método, sin necesidad de *addAttribute* (es lo que acabamos de ver en este apartado de formularios)
 - Si está a nivel de método de controlador, se ejecutará este método antes de cualquiera de sus *request* con lo que podemos añadir elementos “globales” a los Model de todas las peticiones de ese controlador.
- Ejemplo

```
@ModelAttribute
public void nombreMetodo (Model model) {
    model.addAttribute("msg", "Hola mundo!");
}
```

En el ejemplo anterior, sea cual sea el request del controlador, todos los Model pasados a las vistas tendrán el atributo *msg* con el valor “*Hola mundo*”.

Campos de formulario

Hasta este momento solo hemos utilizado campos de texto para la entrada de datos en el formulario para su vinculación con el *command Object*. Podemos emplear cualquier otro tipo de entrada de formularios. Vamos a enumerar los más importantes incluyendo la caja de texto ya vista:

Caja de texto (TextBox):

```
<label>Nombre:<input type="text" id="nombre" th:field="*{nombre}" /></label><br/>
```

Siendo **{nombre}* un atributo del objeto asociado. En principio de tipo String, aunque puede tomar otros valores como por ejemplo numéricos o fechas. En el caso de estas últimas, es aconsejable añadirle al atributo en el objeto la siguiente anotación para especificar el formato y evitar errores.

```
@DateTimeFormat(pattern = "yyyy-MM-dd")
private LocalDate fechaNacimiento;
```

Cuando incorporamos etiquetas Thymeleaf, no funcionan bien los valores por defecto HTML en las cajas de texto (etiqueta *'value'*) por lo que, para estos casos, una opción es eliminar *th:field* y añadir el atributo HTML *name*, asumiendo este la función de *th:field*.

```
<input type="text" name="nombre" value="valor por defecto"/>
```

Botón de chequeo (CheckBox):

```
<label>Acepto:<input type="checkbox" id="acepto" th:field="*{acepto}" /></label><br/>
```

Siendo **{acepto}* un atributo del *commandObject*, de tipo boolean. Tomará valores true o false dependiendo de si se marca o no el check.

Botón de radio (RadioButton):

```
<label><input type="radio" name="button1" value="1" th:field="*{estCivil}">Soltero</label>
<label><input type="radio" name="button1" value="2" th:field="*{estCivil}">Casado</label>
<label><input type="radio" name="button1" value="3" th:field="*{estCivil}">Otro</label>
```

Siendo `*{estCivil}` un atributo del `commandObject`, de tipo String, Long, etc. Tomará el valor indicado en el atributo `value` de la etiqueta HTML. En la mayor parte de los casos, los distintos valores de los botones de opción provendrán del servidor, por ejemplo, de una enumeración o de una colección. Ejemplo:

```
public enum MyEnum { VAL1, VAL2, VAL3 }; //en paquete com.example.demo
```

Esta sería la estructura a aplicar en el formulario para generar dinámicamente un botón de radio para cada uno de los elementos de la enumeración:

```
<div th:each="elem : ${T(com.example.demo.MyEnum).values()}">
  <input type="radio" name="button1" th:value="${elem}" th:text="${elem}"
    th:field="*{myEnum}" />
</div>
```

Siendo `*{myEnum}` un atributo del `commandObject` de tipo de la enumeración que tomará como valor el elemento de la enumeración seleccionado.

- La enumeración debe ir con el nombre de su paquete
- La `T` antes de la enumeración es un operador Thymeleaf

En el caso de ser una lista del modelo y no una enumeración, cambiaríamos solo la primera línea:

```
<div th:each=" elem: ${myList}">
```

Listas desplegadas/desplegables (ListBox, DropdownBox):

```
<select name="curso" size="3" th:field="*{curso}">
  <option value="1">Primero</option>
  <option value="2">Segundo</option>
  <option value="3">Prácticas</option>
</select>
```

Siendo `*{curso}` un atributo del `commandObject`, de tipo String, Long, etc. Tomará el valor indicado en el atributo `value` de la etiqueta HTML. El atributo `size` mayor que uno, determina que sea una lista desplegable o desplegada. Como en el caso de los botones de radio podemos llenarla con valores dinámicos tomados del servidor.

En el ejemplo siguiente, la vista recibiría a través del `Model` una lista llamada `listaProvincias` (por ejemplo, un `ArrayList` de provincias, con atributos `'id'` y `'nombre'`) y cada elemento se introduciría como una opción, tanto en el valor como en el texto a mostrar.

```
<select name="list1" th:field="*{myProvincia}"> <!--atributo en el command obj.-->
  <option value="0">select option</option> <!--opcion por defecto-->
  <option th:each="provin:${listaProvincias}"
    th:value="${provin.id}" th:text="${provin.name}">
  </option>
</select>
```

Análogamente a lo comentado en el caso de las cajas de texto, cuando incorporamos etiquetas Thymeleaf, no funcionan bien los valores por defecto HTML en las listas (etiqueta `'selected'` en el `'option'`) por lo que, para estos casos, una opción es eliminar `th:field` y añadir el atributo HTML `'name'`, asumiendo este la función de `th:field`.

```
<select name="curso">
  <option value="1">Primero</option>
  <option value="2">Segundo</option>
  <option value="3" selected>Prácticas</option>
</select>
```

Si lo que queremos en la lista son los valores de una enumeración, deberemos referenciarla de la misma forma que hacíamos en los botones de opción:

```
<select id="myEnum" th:field="*{myEnum}">
    <option value="">Seleccione un valor</option>           <!--opcional-->
    <option th:each="elem : ${T(com.example.demo.MyEnum).values()}">
        th:value="${elem}" th:text="${elem}"</option>
</select>
```

Formularios de edición

Estos formularios permiten editar campos que ya tienen un valor asignado previamente, en vez de estar vacíos como era el caso anterior. Estos son los típicos formularios de actualización de elementos de nuestro modelo introducidos previamente (updates). La forma de trabajar con ellos es muy similar al caso anterior, pero en este caso el controlador no le pasa a la vista un *commandObject* vacío, sino que tiene los datos del elemento que queremos modificar.

Sería el mismo proceso que para un formulario vacío, pero en el paso 1, donde hacíamos:

```
model.addAttribute ("formInfo", new FormInfo());
```

haríamos:

```
FormInfo formInfo = new FormInfo();
formInfo.setNombre("Ana");
formInfo.setFechaNacimiento(LocalDate.of(2000,01,01));
model.addAttribute ("formInfo ", formInfo);
```

Validación de formularios

Spring nos ofrece un objeto llamado *BindingResult* que combinado con la anotación *@Valid* permite validar de forma automática que los valores enviados en el formulario coinciden con los tipos de datos definidos en el objeto asociado al formulario (al que llamamos *Command Object*).

Adicionalmente ofrece anotaciones que podemos incorporar en la clase del *Command Object* para limitar aún más los valores permitidos en sus atributos. En este caso los usaremos para validar los datos enviados en el formulario, pero podríamos emplearlos en cualquier clase Java.

Esas anotaciones las proporciona Hibernate y preceden a cada atributo. Las más empleadas son:

- *@NotNull*: el atributo no puede ser null.
- *@Min(n)*, *@Max(m)*: especifica un valor mínimo / máximo para el atributo.
- *@NotEmpty*: no puede estar vacío (solo para **String**, Colecciones, Arrays...)
- *@Email*: debe tener formato de email válido.
- *@Size(min = 5, max = 11)*: debe tener un tamaño que cumpla los requisitos.
- *@Past* (una fecha en el pasado). También: *@PastOrPresent*, *@Future*
- *@AssertTrue*: útil por ejemplo para checkbox que deben estar marcados obligatoriamente.

Ejemplo:

```
public class Empleado {
    @Min(0)
    private Long id;
    @NotEmpty
    private String nombre;
    @Email
    private String email;
    @Past
    @DateTimeFormat(pattern = "yyyy-MM-dd")
    private LocalDate fechaNacimiento;
    private double salario;
}
```

Lista completa:

<https://jakarta.ee/specifications/bean-validation/3.0/apidocs/jakarta/validation/constraints/package-summary>

Si Visual Studio Code no encuentra los imports (`jakarta.validation.constraints`), es porque en el pom.xml te falta la dependencia:

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

Una vez modificada la clase que forma el `commandObject`, deberemos cambiar el método que procesa el submit en el controlador (`@PostMapping`) para que valide si lo introducido en los campos de formulario cumplen las restricciones que hemos definido. Para ello:

1.- Añadimos `@Valid` al objeto que recibe los datos del formulario (con o sin `@ModelAttribute`):

```
@PostMapping("/myForm/submit")
public String myformProcessSubmit
    (@Valid FormInfo formInfo, Model model) {
```

2.- Añadimos un nuevo parámetro al método, de tipo `BindingResult`, que nos valdrá para validar el formulario. Este parámetro debe ir justo después del `commandObject` que lleva `@Valid`, si no, no funciona correctamente.

```
@PostMapping("/myForm/submit")
public String myformProcessSubmit
    (@Valid FormInfo formInfo, BindingResult bindingResult, Model model) {
```

3.- Añadir la validación en sí, mediante el método `hasErrors()` de `BindingResult`:

```
@PostMapping("/myForm/submit")
public String myformProcessSubmit
    (@Valid FormInfo formInfo,
     BindingResult bindingResult,
     Model model) {
    if (bindingResult.hasErrors()) return "errorView";
    else return "myFormProcessedView";
}
```

Si no añadimos ninguna anotación en la clase de `command Object`, el `bindingResult` solo validará que los valores introducidos se correspondan con los tipos de la clase, es decir, si es un campo de fecha, que sea una fecha válida, si es un número entero, que no contenga letras ni decimales, etc...

A tener en cuenta:

- Si a un atributo lo anotamos con `@Min(3)` o similares, no se produce error si no se le asigna ningún valor, para ello hay que añadirle también `@NotNull`.
- Si en un formulario, cuando el usuario deja una caja de texto sin cubrir, queremos que `BindingResult` produzca un error debemos incluir `@NotEmpty` y no `@NotNull`.

4.- En la plantilla del formulario, podemos añadir código Thymeleaf para informar de los errores encontrados. `th:classappend` evalúa una expresión y en caso de que se cumpla, añade una clase CSS a la etiqueta, combinada con `#fields.hasErrors` sobre un atributo del `commandObject`, conseguimos que si hay un error cambiar la apariencia de esa etiqueta HTML.

El siguiente ejemplo verificamos que el email introducido cumpla el requisito de formato de email que establecimos en la clase Empleado, que es el `commandObject` de este formulario. Si no lo cumple, aplicará al elemento el estilo css que indiquemos (en este caso 'cssError')

```
<div th:classappend="#{fields.hasErrors('email')} ? 'cssError'">
    <label>Email:<input type="text" id="email" th:field="*{email}" />
</div>
```

Para que este ejemplo funcione, tendríamos que definir en un archivo aparte o en el head el estilo `cssError` (o emplear un estilo de errores externo, por ejemplo 'has-error' de BootStrap).

```
<style>.cssError{ background-color:red; }</style>
```

También podemos añadir un mensaje de error con el atributo Thymeleaf **th:errors**. Este mensaje se define en la clase, con la anotación de validación. En nuestro caso, en la clase Empleado, haríamos:

```
@Email(message = "Debe tener formato email valido")
private String email;
```

y en el formulario, además del código anterior para el formato, añadiríamos un **** para que mostrarse el mensaje.

```
<div th:klassappend="${#fields.hasErrors('email')} ? 'cssError'">
    <label>Email:<input type="text" id="email" th:field="*{email}" />
    <span th:if="${#fields.hasErrors('email')}"
        th:errors="*{email}" class="cssError">texto error</span>
    </label>
</div>
```

Podríamos repetir este proceso para todos los campos del formulario. Hay que señalar que los mensajes de error pueden ir en “hardcode” en la clase, pero también se podría rescatar desde un fichero de propiedades.

Subida de ficheros

En nuestros formularios podemos querer anexar archivos para ser enviados al servidor. Esto lo haremos mediante la propiedad ‘multipart’ del protocolo HTTP. Entendemos por ‘multipart’ aquel mensaje que tiene diferentes secciones o partes, y que cada una de ellas puede tener un tipo de contenido diferente de forma que en una respuesta podríamos enviar: texto plano, ficheros de texto, una imagen, etc. Cada parte vendrá delimitada por una línea con el atributo “Content-type” que indicará el tipo de contenido de esa parte.

Los pasos a seguir serán:

- 1.- Añadir a la etiqueta **<form>** del formulario de subida el atributo: **enctype="multipart/form-data"**.
- 2.- El formulario incluirá un campo para la subida de ficheros:

```
<label>Adjuntar un fichero:<input type="file" name="file"/></label><br/>
```

3.- En el controlador, añadir en la firma del método que procesa el formulario (@PostMapping) un nuevo parámetro: **@RequestParam MultipartFile file**, siendo “file” el nombre que le hemos dado en el formulario al campo para la subida de ficheros. La clase **MultipartFile** nos permitirá, mediante sus métodos, procesar el fichero recibido: *getName()*, *isEmpty()*, etc.

4.- En el método del punto anterior, añadiremos la lógica que necesitemos aplicar al fichero, probablemente en un servicio:

```
@PostMapping("/myForm/submit")
public String processForm( FormInfo formInfo, @RequestParam MultipartFile file) {
    myService.tratarFichero(file);
    . . .
```

Lo habitual es almacenarlo en nuestro sistema, como veremos en el apartado siguiente.

5.- No sería necesario modificar el *commandObject* del formulario ya que el fichero viene en un parámetro aparte, pero en muchos casos sí querremos almacenar la ruta donde hemos almacenado el fichero. Si el *commandObject* es una clase de nuestro modelo (Cliente, Producto, etc.) es frecuente añadir a esa clase un atributo nuevo de tipo String para almacenar la ruta local del fichero subido.

Almacenamiento de ficheros

Vamos a ver como almacenar los ficheros que suban los usuarios mediante un formulario. En el mundo real probablemente usásemos un servidor externo para el almacenamiento de ficheros, en nuestro caso, para simplificarlo, se guardarán en el propio servidor.

Tenemos que modificar el controlador, pero necesitamos un servicio que implemente las operaciones sobre el fichero. Si siguiésemos la metodología mostrada en el tema dedicado a servicios definiríamos una interfaz con los métodos necesarios y luego una clase que los implementase; vamos a obviar esta propuesta, y hacer una clase lo más sencilla posible.

El servicio tendría un atributo que representaría la ruta donde almacenaríamos los archivos y tres métodos: uno para guardar el archivo (cuando se envía el formulario), otro para recuperarlo (cuando queremos mostrarlo, por ejemplo) y uno de borrado para eliminarlo cuando ya no sea necesario:

```
import java.io.IOException;
import java.io.InputStream;
import java.nio.file.Files;
import java.nio.file.Path;
import java.nio.file.Paths;
import java.nio.file.StandardCopyOption;

import org.springframework.core.io.Resource;
import org.springframework.core.io.UrlResource;
import org.springframework.stereotype.Service;
import org.springframework.util.StringUtils;
import org.springframework.web.multipart.MultipartFile;

@Service
public class FileStorageService {
    private final Path rootLocation = Paths.get("uploadDir");

    public String store(MultipartFile file) throws RuntimeException {
        if (file.isEmpty()) throw new RuntimeException("Fichero vacío");
        String filename = StringUtils.cleanPath(file.getOriginalFilename());
        if (filename.contains("..")) { throw new RuntimeException("Fichero incorrecto");}
        String extension = StringUtils.getFilenameExtension(filename);
        String storedFilename = System.currentTimeMillis() + "." + extension;

        try (InputStream inputStream = file.getInputStream()) {
            Files.copy(inputStream, this.rootLocation.resolve(storedFilename),
                    StandardCopyOption.REPLACE_EXISTING);
            return storedFilename;
        } catch (IOException ioe) {throw new RuntimeException("Error en escritura");}
    }

    public void delete(String filename) throws RuntimeException {
        try {
            Path file = rootLocation.resolve(filename);
            if (!Files.exists(file)) throw new RuntimeException("No existe el fichero");
            Files.delete(file);
        } catch (IOException ioe) { throw new RuntimeException("Error en borrado");}
    }

    public Resource loadAsResource(String filename) throws RuntimeException {
        try { Path file = rootLocation.resolve(filename);
            Resource resource = new UrlResource(file.toUri());
            if (resource.exists() || resource.isReadable()) return resource;
            else throw new RuntimeException("Error IO");
        } catch (Exception e) { throw new RuntimeException("Error IO");}
    }
}
```

La carpeta en la que almacenaremos los archivos, en este ejemplo 'uploadDir' debe crearse previamente y estará situada en la ruta raíz del proyecto cuando lo ejecutamos desde el IDE y en la misma carpeta que el "jar" cuando la aplicación esté desplegada.

En el controlador inyectaríamos este nuevo servicio creado:

```
@Controller
public class MyController {
    @Autowired
    private FileStorageService fileStorageService;
```

y en el método que recibe el submit del formulario añadimos lo que comentamos en el apartado anterior: `@RequestParam MultipartFile file`:

```
@PostMapping("/myForm/submit")
public String showMyFormSubmit(FormData formData, @RequestParam MultipartFile file) {
    try {
        String newFileName = fileStorageService.store(file);
        //devuelve el nombre definitivo con el que se ha guardado.
        //Lo podríamos guardar en algún objeto
        System.out.println("Fichero almacenado: " + newFileName);
    } catch (Exception e) { return "redirect:/myForm?err=1"; }
    //resto de tareas del controlador. Por ejemplo: myService.save(mco);
```

Si en alguna vista queremos mostrar un fichero guardado en el servidor, por ejemplo, una imagen, añadiríamos una etiqueta como esta:

```

```

Siendo `${imagen}` una variable que contendrá el nombre con la que esté guardada la imagen. Finalmente, en el controlador tendremos que receptionar esa ruta y devolver al navegador la imagen:

```
@GetMapping("/files/{filename:.+}")
public ResponseEntity<Resource> serveFile(@PathVariable String filename) {
    Resource file = fileStorageService.loadAsResource(filename);
    return ResponseEntity.ok().body(file);
}
```

Podemos limitar el tamaño de los archivos que permitimos subir al servidor mediante estas dos propiedades en el archivo `application.properties`:

```
spring.servlet.multipart.max-file-size=256KB
spring.servlet.multipart.max-request-size=256KB
```

Si se sobrepasa el límite se producirá una `java.lang.IllegalStateException` que será necesario tratar.

Importante: En los recursos disponibles en el área de descargas hay un ejemplo completo de un CRUD sobre base de datos de una entidad sencilla llamada Cliente. En el ejemplo se muestra como se gestionan la subida de archivos, en este caso cada cliente tendrá asignada una foto. La aplicación permite: en el alta/edición de un cliente asignarle una foto; en el borrado de cliente eliminar su foto y al consultar los datos de los clientes mostrar la foto asignada a cada uno. Al acabar el tema 7 (base de datos), puedes consultar el ejemplo para comprender mejor este apartado.

Envío de emails

Será bastante habitual que nuestras aplicaciones necesiten enviar emails, tanto a los usuarios externos como a los administradores de la aplicación cuando se produzcan ciertos eventos. Ejemplos típicos pueden ser formularios de contacto, la recepción de un pedido, una notificación de baja de cliente, etc.

Una opción para enviar emails sería la instalación de un servidor de correo, pero sería bastante compleja; tenemos una opción más sencilla que consiste en usar a Gmail como reenviador de emails, es decir, en vez de enviar nosotros el email, le diremos a Gmail que lo haga por nosotros. Como

requisito solo precisamos tener una cuenta de Google con la verificación en dos pasos activada y que disponga de una contraseña de aplicaciones: tienes más información en este enlace sobre este proceso: <https://support.google.com/accounts/answer/185833?hl=es>

No uses tu cuenta de correo personal, crea una solo para esta tarea, ya que la contraseña de aplicaciones quedará visible en el fichero `application.properties`.

El proceso es muy sencillo, ya que simplemente debemos invocar una función a la que le pasamos como parámetro el destinatario, asunto del mensaje, cuerpo del mensaje y, opcionalmente, archivos adjuntos. Lo habitual será crear una clase de servicio que incluya esta función, y que podrá ser invocada desde cualquier punto de la aplicación.

Estos son los pasos a seguir:

1. Añadir la dependencia `starter-mail` en el `pom.xml`.

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

2. Configuración del `application.properties`

```
spring.mail.host=smtp.gmail.com
spring.mail.port=587
spring.mail.username=correo de Gmail con verificación en dos pasos activada
spring.mail.password=contraseña de aplicaciones (no la contraseña de usuario)
spring.mail.properties.mail.smtp.starttls.enable=true
spring.mail.properties.mail.smtp.starttls.required=true
spring.mail.properties.mail.smtp.auth=true
spring.mail.properties.mail.smtp.connectiontimeout=5000
spring.mail.properties.mail.smtp.timeout=5000
spring.mail.properties.mail.smtp.writetimeout=5000
```

3. Crear el servicio con método que realiza el envío:

```
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.mail.javamail.JavaMailSender;
import org.springframework.mail.javamail.MimeMessageHelper;
import org.springframework.stereotype.Service;
import jakarta.mail.MessagingException;
import jakarta.mail.internet.MimeMessage;
@Service
public class EmailService {
    @Autowired
    private JavaMailSender sender;

    public boolean sendEmail(String destination, String subject, String textMessage) {
        try {
            MimeMessage message = sender.createMimeMessage();
            MimeMessageHelper helper = new MimeMessageHelper(message);
            helper.setTo(destination);
            helper.setText(textMessage, true);
            helper.setSubject(subject);
            sender.send(message);
            return true;
        } catch (MessagingException e) { e.printStackTrace(); return false; }
    }
}
```

4. Podemos llamar a este método desde cualquier punto de la aplicación, inyectando el servicio EmailService y llamando al método anterior. El cuerpo del mensaje admite etiquetas HTML que se reflejarán en el email recibido.

```
@Autowired
private EmailService emailService;
String cuerpoMensaje = "<h1>Cuerpo del mensaje</h1>";
String destinatario= "rdf@fernandowirtz.com"
String asunto = "Asunto del mensaje";
boolean envioEmail = emailService.sendEmail(destinatario,asunto,cuerpoMensaje);
if (envioEmail) . . .
else . . .
```

Podemos modificar el método de servicio para poder enviar a varios destinatarios (ya que el método *helper.setTo* puede recibir un String para un destinatario o bien un String[] para varios)

```
public boolean sendEmail(String [] destinations, String subject, String textMessage) {
    try {
        MimeMessage message = sender.createMimeMessage();
        MimeMessageHelper helper = new MimeMessageHelper(message);
        helper.setTo(destinations);
        . . .
    }
```

Y la llamada la modificaríamos así:

```
String [] destinatarios= new String[] {"rdf@fernandowirtz.com","zzz@fernandowirtz.com"};
boolean envioEmail = emailService.sendEmail(destinatarios, asunto, cuerpoMensaje);
```

5. Podemos también enviar uno o más ficheros adjuntos, para ello, pasaremos como parámetro al método el nombre del archivo (o un array con los nombres de los archivos) a enviar y añadiremos un segundo parámetro al objeto MimeMessageHelper con valor *true*. Finalmente, adjuntaremos el o los archivos mediante el método *addAttachment*. Este sería el código para un archivo:

```
public boolean sendEmail(String destination, String subject, String textMessage,
    String attachment) {
    try {
        MimeMessage message = sender.createMimeMessage();
        MimeMessageHelper helper = new MimeMessageHelper(message, true);
        helper.setTo(destination);
        helper.setText(textMessage, true);
        helper.setSubject(subject);
        File archivoAdjunto = new File(attachment);
        helper.addAttachment(archivoAdjunto.getName(), archivoAdjunto);
        sender.send(message);
        return true;
    } catch (MessagingException e) { e.printStackTrace(); return false; }
}
}
```

Y este para varios archivos adjuntos:

```
public boolean sendEmail(String destination, String subject, String textMessage,
    String[] attachments) {
    try {
        MimeMessage message = sender.createMimeMessage();
        MimeMessageHelper helper = new MimeMessageHelper(message, true);
        helper.setTo(destination);
        helper.setText(textMessage, true);
        helper.setSubject(subject);
        for (String attachment : attachments) {
            File archivoAdjunto = new File(attachment);
            helper.addAttachment(archivoAdjunto.getName(), archivoAdjunto);
        }
        . . .
    }
```

Y la llamada sería así:

```
String archivo= "myFiles/img.png";
boolean envioEmail=emailService.sendEmail(destinatario,asunto,cuerpoMensaje, archivo);
if (envioEmailOk) . . .
```

O bien:

```
String [] archivo= new String[] { "myFiles/img.png", "myFiles/img2.png" };
boolean envioEmail=emailService.sendEmail(destinatario,asunto,cuerpoMensaje, archivos);
if (envioEmailOk) . . .
```

siendo *myfiles* una carpeta ubicada en la raíz del proyecto.

Ejecución asíncrona

Tal y como lo hemos planteado, el envío del correo puede tardar unos segundos. Si no queremos que el navegador del cliente se quede a la espera de que el envío sea efectivo, podemos hacer un envío asíncrono. De esta forma, se devolverá el control a la línea de código posterior al envío de correo de forma inmediata, sin espera alguna. En este caso, no podremos confirmarle al cliente si el correo se ha enviado correctamente o no. Para hacer una llamada asíncrona deberemos hacer las siguientes modificaciones en nuestro código:

- Añadir a la clase main la anotación: `@EnableAsync` (además de `@SpringBootApplication`)
- Añadir al método que hace el envío de email la anotación: `@Async`

```
@Async
public void sendEmail(String destination, String subject, String textMessage) {
```

- Este método deberá devolver *void* ya que no se esperará a su finalización y por tanto no puede devolver nada.

Tema 6: Modelo y Repositorios

Entidades

El modelo de dominio de nuestro sistema va a ser el conjunto de toda la información relevante para la aplicación. Cada elemento de ese esquema se denomina **entidad** cada entidad tendrá características que la describen y que tomarán diferentes valores para cada instancia de entidad. Por otra parte, esas entidades tendrán relaciones y asociaciones entre ellas.

En el caso de la programación orientada a objetos esas entidades serán clases y sus características serán los atributos de cada clase. Finalmente, en un ambiente relacional, esas clases serán mapeadas a tablas y los atributos a columnas de las tablas, para lograr su persistencia, aunque eso lo veremos en el tema siguiente.

Las clases que son las entidades de nuestro modelo clases incluirán generalmente los atributos con acceso privado, getters, setters, constructores, métodos comunes (equals, hashCode, toString, etc) y métodos propios de su comportamiento.

Como vimos en el apartado de formularios, adicionalmente las clases pueden incorporar ciertas restricciones en sus atributos, que serán validadas de forma automática por el sistema, produciendo una excepción en el caso de que no se cumplan. Ya las comentamos en la validación de formularios, pero recordamos estas:

- `@Min (value=0, message="{empleado.id.mayorquecero}")`
- `@NotEmpty`
- `@Email`

En cuanto a la ubicación de las clases que forma el modelo de dominio, podemos hacer un paquete con todas ellas y llamarle por ejemplo **domain**, o bien crear paquetes orientados a cada dominio, y que un mismo paquete (Cliente, Producto, etc.) contenga su controlador, servicio, ...Como siempre estos paquetes deben ser subpaquetes del paquete raíz que contiene la app (la clase con el main).

*Los command Object de los formularios, si son hechos ad-hoc y no se corresponden con ninguna entidad del dominio, podemos meterlos en una carpeta/paquete llamada **dto** (Data Transfer Objects), hablaremos de ellos en el capítulo siguiente.*

Lombok

Lombok es una librería que, a través de anotaciones, reduce el código común que tenemos que codificar ahorrándonos tiempo y mejorando la legibilidad del mismo. Con esas anotaciones se pueden generar de forma automática getters, setters, constructores, etc. y esas transformaciones en el código se realizan en tiempo de compilación.

Tiene multitud de anotaciones, que se pueden emplear a nivel atributo, método, clase, etc. Estas serían algunas de las más utilizadas:

- **@Getter**: genera getter público para el atributo. Si lo podemos antes de la clase, lo genera para todos sus atributos. Los getters comienzan por 'get' salvo para atributos de tipo boolean que comienzan por 'is'
- **@Setter**: Análogo a getter, pero generando setters.
- **@EqualsAndHashCode**: Genera los métodos equals y hashCode de la clase. Por defecto, ambos métodos se basarán en todos los atributos de la clase, aunque se puede modificar este comportamiento como veremos a continuación.
- **@ToString**: Genera una cadena con el nombre de la clase y con cada atributo y su valor separado por comas.

- **@NoArgsConstructor**: genera el constructor por defecto.
- **@RequiredArgsConstructor**: genera el constructor con parámetros finales o marcados con **@NonNull**
- **@AllArgsConstructor**: genera el constructor con parámetros para todos los atributos.
- **@Data**: es de los más utilizados y agrupa las anotaciones: Getter, Setter, ToString, EqualsAndHashCode y RequiredArgsConstructor.
- **@Builder**: genera un método para instanciar la clase de una forma más legible que con un constructor y desacopla dicha instancia, de forma que, aunque en un futuro cambien los constructores de la clase, la instancia con *builder* seguirá funcionando.

Estas anotaciones y muchas otras son parametrizables, de forma que podemos adaptar su comportamiento a nuestras necesidades. Por ejemplo, si queremos que los métodos *equals* y *hashCode* solo tengan en cuenta un atributo para la comparación, podemos parametrizarlo así:

`@EqualsAndHashCode (of="atributo")`

Para emplear Lombok en proyectos Maven solo debemos buscar su dependencia en el repositorio oficial: <https://mvnrepository.com/> e incluir su dependencia en el pom.xml de nuestro proyecto.

Project Lombok > 1.18.24

Spice up your java: Automatic Resource Management, automatic generation of getters, setters, equals, hashCode and toString, and more!

License	MIT
Categories	Code Generators
HomePage	https://projectlombok.org
Date	(Apr 18, 2022)
Files	pom (1 KB) jar (1.9 MB) View All
Repositories	Central
Used By	15,754 artifacts

Maven **Gradle** **Gradle (Short)** **Gradle (Kotlin)** **SBT** **Ivy** **Grape** **Leiningen**
Buildr

```
<!-- https://mvnrepository.com/artifact/org.projectlombok/lombok -->
<dependency>
  <groupId>org.projectlombok</groupId>
  <artifactId>lombok</artifactId>
  <version>1.18.24</version>
  <scope>provided</scope>
</dependency>
```

Include comment with link to declaration

Ahora, en la clase que deseemos, añadiremos las anotaciones que generarán el código, por ejemplo:

```

1 package com.mycompany.mavenproject4;
2
3 import lombok.AllArgsConstructor;
4 import lombok.Builder;
5 import lombok.Data;
6 import lombok.EqualsAndHashCode;
7 import lombok.NoArgsConstructor;
8
9
10 @Data
11 @AllArgsConstructor
12 @NoArgsConstructor
13 @EqualsAndHashCode (of="nombre")
14 // @EqualsAndHashCode (of = {"nombre", "edad"})
15 @Builder
16 public class Alumno {
17     private String nombre;
18     private int edad;
19     private boolean graduado;
20 }
```

En este caso se generarían getters, setters, `toString`, el constructor con todos los parámetros, el constructor sin parámetros, y los métodos `equals` y `hashCode` basados solo en el atributo nombre. También el método `builder` que comentamos antes.

Ahora podemos emplear todo el código generado como si lo hubiésemos escrito nosotros mismos:

```
Alumno a1 = new Alumno("Pepe", 13, false);
Alumno a2 = Alumno.builder()
    .nombre("Pepe")
    .edad(12)
    .build();
a2.setGraduado(true);
if (a1.equals(a2)) System.out.println(a1.toString())
```

Logging

El sistema de logging de nuestra aplicación se encarga de mostrar los distintos eventos que están ocurriendo durante la ejecución. Con los logs podemos descubrir errores, comportamientos extraños, pero también auditar ataques. En Java hay varios sistemas de logging (Log4j, logback, java.util.logging) y existe una capa de abstracción sobre todos ellos: slf4j. En el tema 10 se estudian los logs en detalle.

En estas librerías de logging las trazas se emiten a través de un logger que normalmente se corresponde con el nombre de la clase en la que se emite la traza. De esta forma las trazas se pueden filtrar por el nivel de importancia de la traza (debug, info, warn, ...) y por el nombre del logger de forma que podemos obtener un registro de las trazas emitidas por los loggers que deseamos.

Si añadimos la anotación Lombok `@Slf4j` en nuestra clase, se generará un log así:

```
private static final org.slf4j.Logger log = org.slf4j.LoggerFactory.getLogger(LogExample.class);
```

Una vez incorporada, podemos usar sus métodos para añadir información al log con distintos niveles de criticidad (info, warn, error...) y que veremos por consola.

```
log.info ("Información al log. . . ");
```

Eliminando `@Autowired` con Lombok:

Desde la versión 4.3 de Spring, si a una clase le añadimos la anotación `@RequiredArgsConstructor` y a los elementos que queremos “autocablear” les añadimos el atributo `final`, ya no es necesario añadir la anotación `@Autowired`. Por ejemplo:

```
@Controller
public class EmpleadoController {
    @Autowired
    private EmpleadoService empleadoService;
```

Sería equivalente a:

```
@Controller
@RequiredArgsConstructor
public class EmpleadoController {
    private final EmpleadoService empleadoService;
```

Repositorios

Un **repository** es una capa adicional de nuestra aplicación que permite gestionar los datos almacenados en alguna colección. Generalmente la capa de servicio (o directamente el controlador) incluirá un repositorio e invocará a sus métodos que harán el acceso a los datos. Así pues, la capa de repositorio actuará sobre el modelo y tendrá métodos para añadir a la colección, modificar, eliminar, consultar, etc.

En principio, en este capítulo, vamos a ver repositorios creados por nosotros mismos y en memoria y en el siguiente capítulo veremos los repositorios con persistencia, sobre una base de datos, y comprobaremos como Spring hace prácticamente todo el “trabajo sucio” por nosotros.

Así pues, si estamos trabajando con la clase *Producto*, un repositorio sería algo tan sencillo como definir una lista (interfaz *List*) instanciada con un *ArrayList* de *Producto*.

La interfaz *List* cumple su función como repositorio ya que tiene definidos métodos como *get()*, *add()*, *remove()*, *set()*, etc. La clase *ArrayList* implementa dichos métodos.

Este esquema se repetirá en el capítulo siguiente a nivel base de datos, las interfaces *JpaRepository* o *CRUDRepository* definirán métodos para acceder a nuestros datos en la base de datos e *Hibernate* implementará dichos métodos para acceder a la base de datos. Nosotros solo configuraremos y los invocaremos.

La forma de encajar el repositorio en nuestra aplicación es incluyéndolo en nuestro servicio. En el caso un repositorio en memoria podría ser algo como:

```
@Service
public class ProductoService {
    private List<Producto> repositorio = new ArrayList<>();
    ...
}
```

Y definiendo las operaciones que queremos implementar sobre el mismo. Las más básicas serían las referidas habitualmente con el acrónimo CRUD (Create, Read, Update, Delete), además de alguna otra como listar todos los elementos de repositorio, búsquedas, etc.

Por ejemplo, para añadir un elemento al repositorio (al *ArrayList* en este caso):

```
public void addProducto (Producto producto) {
    repositorio.add(producto);
    //podría devolver boolean en caso de que no siempre se añadiese
}
```

Y de forma análoga el resto de métodos que precisemos como vamos a ver en el siguiente apartado.

CRUD con Repositorios en Memoria

Con lo que hemos visto hasta ahora ya podemos hacer una aplicación completa que sea capaz de gestionar una entidad de nuestro modelo, con las operaciones típicas sobre el mismo. Vamos a verlo paso a paso, con un ejemplo concreto.

Supongamos que queremos gestionar un repositorio de los empleados de una empresa, en este caso con un repositorio en memoria. En el capítulo siguiente lo haremos persistente, mediante un repositorio sobre base de datos. Los pasos a seguir serían:

- 1.- Crear un nuevo proyecto. Dependencias: *starter-web*, *devtools*, *lombok*, *starter-thymeleaf* y *starter-validation*. Opcionalmente *webjars-bootstrap* y *webjars-locator*.
- 2.- Crear la estructura de carpetas/paquete que precisemos: podemos optar por carpetas: *domain*, *controllers* y *services* mejor que todos los elementos una única carpeta.
- 3.- Crear las clases que conforman nuestro modelo de dominio. En este caso sería solo la clase *Empleado*. Podemos ayudarnos de Lombok para escribir menos código.

```
@Data
@AllArgsConstructor
@NoArgsConstructor
@EqualsAndHashCode(of = "id")

public class Empleado {
    @Min(value = 0)
    private Long id;

    @NotEmpty
    private String nombre;

    @Email (message = "Debe tener formato email valido")
    private String email;

    private Double salario;

    private boolean enActivo;

    private Genero genero;
}
```

Y creamos también la enumeración para el género, dentro de la carpeta *domain*:

```
package com.example.myapp.domain;
public enum Genero { MASCULINO, FEMENINO, OTROS };
```

4.- Crear el servicio que contiene el repositorio con los métodos CRUD del servicio que invocan finalmente a los métodos del repositorio (en este caso de *List*)

```
@Service
public class EmpleadoService {
    private List<Empleado> repositorio = new ArrayList<>();

    public List<Empleado> obtenerTodos() {
        return repositorio;
    }

    public Empleado obtenerPorId(long id) {
        for (Empleado empleado : repositorio)
            if (empleado.getId() == id)
                return empleado;
        return null;                                //podría lanzar excepción si no encontrado
    }

    public Empleado añadir (Empleado empleado) {
        if (repositorio.contains(empleado)) return null; //ver equals Empleado (mismo id)
        repositorio.add(empleado);                    //podría no devolver nada, o boolean, etc.
        return empleado;
    }

    public Empleado editar (Empleado empleado) {
        int pos = repositorio.indexOf(empleado);
        if (pos == -1) return null;                  //debería lanzar excepción si no encontrado
        repositorio.set(pos, empleado);            //si lo encuentra, lo sustituye
        return empleado;
    }

    public void borrar (Long id) {
        Empleado empleado = this.obtenerPorId(id);
        if (empleado != null) {
            repositorio.remove(empleado);          //podría devolver boolean
        }
    }
}
```

En el código expuesto no se contemplan excepciones, devolviendo valores *null* al controlador en caso de que exista algún error en tiempo de ejecución. Una solución más profesional, como comentamos en capítulos previos, sería que el servicio lanzase excepciones; bien creadas por nosotros, bien de tipo *RuntimeException*. Ejemplo:

```
public Empleado editar(Empleado empleado) throws RuntimeException{
    int pos = repositorio.indexOf(empleado);
    if (pos == -1)
        throw new RuntimeException ("Empleado no encontrado");
    repositorio.set(pos, empleado);
    return empleado;
}
```

El controlador sería el encargado de capturarlas mediante *try...catch* y pasar el mensaje de excepción a las vistas tal y como comentamos en el tema 4.

5.- Crear el controlador que recibe las peticiones del usuario e invoca al servicio adecuado. El servicio está inyectado (@Autowired) en el controlador. Para el alta de un nuevo empleado o para editar un empleado existente necesitaremos formularios (@PostMapping). Las URL podrían ser las siguientes:

/list (o simplemente /)	: listar los datos de todos los empleados
/{id}	: obtener los datos solo del empleado 'id'
/nuevo	: formulario para añadir un nuevo empleado
/editar/{id}	: formulario de edición para modificar empleado 'id'
/borrar/{id}	: eliminar el empleado 'id'

Y este sería el controlador, suponiendo que el servicio no lanza excepciones, si no que devuelve nulos en caso de error. En caso de que sí devolviese excepciones sustituiríamos los 'if' por 'try...catch'.

```
@Controller
public class EmpleadoController {

    @Autowired
    private EmpleadoService empleadoService;

    @GetMapping({ "/", "/list" })
    public String showList(@RequestParam(required = false) Integer numMsg, Model model) {
        if (numMsg != null) {
            switch(numMsg){
                case 1 -> model.addAttribute("msg", "Empleado no encontrado");
                case 2 -> model.addAttribute("msg", "Formulario incorrecto");
            }
        }
        model.addAttribute("listaEmpleados", empleadoService.obtenerTodos());
        return "listView";
    }

    @GetMapping("/{id}")
    public String showElement(@PathVariable long id, Model model) {
        Empleado empleado = empleadoService.obtenerPorId(id);
        if (empleado == null) return "redirect:/?numMsg=1";
        model.addAttribute("empleado", empleado);
        return "listOneView";
    }

    @GetMapping("/nuevo")
    public String showNew(Model model) {
        //el command object del formulario es una instancia de empleado vacia
        model.addAttribute("empleadoForm", new Empleado());
        return "newFormView";
    }
}
```

```

    @PostMapping("/nuevo/submit")
    public String showNewSubmit(@Valid Empleado empleadoForm,
        BindingResult bindingResult) {
        if (bindingResult.hasErrors()) return "redirect:/?numMsg=2";
        Empleado empleado = empleadoService.agregar(empleadoForm);
        if (empleado == null)
            return "redirect:/?numMsg=2";
        return "redirect:/list";
    }
    @GetMapping("/editar/{id}")
    public String showEditForm (@PathVariable long id, Model model) {
        Empleado empleado = empleadoService.obtenerPorId(id);
        //el commandobject del formulario es el empleado con el id solicitado
        if (empleado == null) return "redirect:/?numMsg=1";//si no encuentra vuelve al inicio.
        model.addAttribute("empleadoForm", empleado);
        return "editFormView";
    }
    @PostMapping("/editar/{id}/submit")
    public String showEditSubmit(@PathVariable Long id, @Valid Empleado empleadoForm,
        BindingResult bindingResult) {
        if (bindingResult.hasErrors()) return "redirect:/?numMsg=2";
        Empleado empleado = empleadoService.editar(empleadoForm);
        if (empleado == null) return "redirect:/?numMsg=2";
        return "redirect:/list";
    }
    @GetMapping("/borrar/{id}")
    public String showDelete(@PathVariable long id) {
        empleadoService.borrar(id);
        return "redirect:/list";
    }
}

```

6.- Crear vistas del cliente:

El listado inicial: */listView.html*

```

<!DOCTYPE html>
<html lang="es" xmlns:th="http://www.thymeleaf.org">
    <head> . . . </head>
    <body>
        <h1>Listado de empleados de la empresa</h1>
        <table><thead>
            <tr><th>ID</th><th>Nombre</th><th>Email</th><th>Salario</th>
            <th>En Activo</th><th>Género</th><th>Editar</th><th>Borrar</th></tr>
        </thead>
        <tbody>
            <tr th:each="empleado : ${listaEmpleados}">
                <td th:text="${empleado.id}">Id</td>
                <td><a th:href="@{/editar/{id}(id=${empleado.id})}"> //el nombre es un enlace
                    th:text="${empleado.nombre}">nombre</a></td>
                <td th:text="${empleado.email}">email@gmail.com</td>
                <td th:text="${empleado.salario}">0</td>
                <td th:text="${empleado.enActivo}">bool</td>
                <td th:text="${empleado.genero}">genero</td>
                <td><a th:href="@{/editar/{id}(id=${empleado.id})}">Editar</a></td>
                <td><a th:href="@{/borrar/{id}(id=${empleado.id})}">Borrar</a></td>
            </tr>
        </tbody></table>
        <p th:if="${msg!=null}">Error: <span th:text="${msg}">err</span></p>
        <a th:href="@{/}">Inicio</a><br/>
        <a th:href="@{/nuevo}">Nuevo empleado</a><br/>
    </body>
</html>

```

El formulario para nuevo empleado, cuyo `<body>` sería:

```

<h1>Nuevo empleado</h1>
<form method="post" action="#" th:action="@{/nuevo/submit}" th:object="${empleadoForm}">
    <label>Id: <input type="text" id="id" th:field="*{id}" /> </label><br/>
    <label>Nombre: <input type="text" id="nombre" th:field="*{nombre}" /></label><br/>
    <label>Email: <input type="text" id="email" th:field="*{email}" /></label><br/>
    <label>Salario: <input type="text" id="salario" th:field="*{salario}" /></label><br/>
    <label>En Activo:<input type="checkbox" id="enActivo" th:field="*{enActivo}" /></label><br/>
    Genero:<br/>
    <div th:each="gen : ${T(com.example.myapp.domain.Genero).values()}">
        <input type="radio" name="button1" th:value="${gen}" th:text="${gen}" th:field="*{genero}" />
    </div>
    <input type="submit" value="Enviar"/>
</form>
<a th:href="@{/}">Inicio</a><br/>

```

Podríamos añadirle tratamiento si no cumple alguno de los requisitos establecidos en la entidad, como por ejemplo, la validación de formato de email. Para ello podemos definir un estilo CSS (o usar uno de BootStrap como 'has-error').

```
<style>.cssError { background-color:red; } </style>
```

Y modificar el campo de formulario para que si hay algún error muestre el texto de error que se estableció en la entidad mediante el atributo `message`, y formatearlo con el estilo CSS creado.

```

<div th:append="${#fields.hasErrors('email') ? 'cssError'">
    <label>Email:<input type="text" id="email" th:field="*{email}" />
        <span th:if="${#fields.hasErrors('email')}" th:errors="*{email}" class="cssError">texto error</span>
    </label>
</div>

```

8.- El formulario de edición de empleado, que sería prácticamente igual al anterior salvo la línea:

```
<form method="post" action="#" th:action="@{/editar/submit}" th:object="${empleadoForm}">
```

O para tener una URL que incluya el `id` del elemento a modificar, `th:action` podría ser:

```
th:action="@{/edit/{id}/submit(id=${empleadoForm.id})}"
```

y al input atributo `id` le añadiríamos el atributo `readonly` ya que no tiene sentido cambiar el identificador único, no sería una modificación, sería crear un nuevo elemento:

```
<label>Id:<input type="text" readonly id="id" th:field="*{id}" /> </label><br/>
```

9.- La vista `listOneView.html` sería muy sencilla, solo mostrando los atributos de un solo empleado.

10.- Podemos meter un `commandLineRunner` para añadir inicialmente algún dato y que no arranque con la lista vacía. Añadiríamos en la clase aplicación (la que contiene el `main` y la anotación `@SpringBootApplication`).

```

@SpringBootApplication
public class EmpleadoApplication {
    public static void main(String[] args) {
        SpringApplication.run(EmpleadoApplication.class, args);
    }
    @Bean
    CommandLineRunner initData(EmpleadoService empleadoService) {
        return args -> {
            empleadoService.añadir(
                new Empleado(1L, "pepe", "pepe@gmail.com", 25000d, true, Genero.MASCULINO));
            empleadoService.añadir(
                new Empleado(2L, "ana", "ana@gmail.com", 28000d, true, Genero.FEMENINO));
        };
    }
}

```

Duda: ¿Esta instancia de *EmpleadoService* es la misma que la que se crea (inyecta) en el controlador? Y aquí es un parámetro de un método ¿Quién lo instancia?

La respuesta a ambas dudas es parte del funcionamiento del Inversor de Control de Spring. El IoC crea una instancia de cada componente bajo su control cuando es necesario, y como lo hace con el patrón Singleton (una misma instancia para todas las peticiones), efectivamente es el mismo servicio en toda la ejecución de la aplicación.

11.- Por último, podríamos crear una interfaz para el servicio y que la clase anterior *EmpleadoService* fuese refactorizada a *EmpleadoServiceImpl* la implementase.

```
public interface EmpleadoService {  
    Empleado añadir (Empleado empleado);  
    List<Empleado> obtenerTodos ();  
    Empleado obtenerPorId (long id);  
    Empleado editar (Empleado empleado);  
    void borrar (Long id);  
}  
@Service  
public class EmpleadoServiceImpl implements EmpleadoService {  
    . . .
```

12.- De forma opcional, podríamos poner en práctica lo visto en el tema de formularios para añadir una imagen al empleado, pero no lo haremos por ahora.

Filtros en la vista

El proceso visto hasta ahora permite realizar un CRUD completo sobre la entidad *Empleado*, pero generalmente necesitaremos filtros para que, cuando consultamos los empleados, podamos filtrar por algún atributo y obtener un subconjunto de los mismos.

Filtro por texto

Este es un filtro típico en el que al usuario se le presenta una caja de texto en la que el usuario introduce un texto y al enviarlo al servidor, filtrará los elementos que coincidan total o parcialmente en un determinado atributo. En general no se distinguen minúsculas y mayúsculas en estos filtros. Vemos un ejemplo de filtro por nombre de empleado:

Vista:

```
<form method="post" action="#" th:action="@{/findByName}" th:object="${findForm}">  
    <label>Buscar por nombre:<input type="text" name="nombre" th:field="*{nombre}"/></label>  
    <input type="submit" value="Buscar"/>  
</form>
```

Al formulario le enviamos un Empleado como *commandObject* pero en realidad podríamos enviarle una clase mucho más sencilla (un DTO) solo con un String para recibir el texto a buscar en el nombre.

Controlador:

```
@PostMapping("/findByName")  
public String showFindByNameSubmit(Empleado empleadoForm, Model model) {  
    model.addAttribute("listaEmpleados",  
        empleadoService.buscarPorNombre(empleadoForm.getNombre()));  
    model.addAttribute("findForm", empleadoForm);  
    return "listView";  
}
```

Servicio:

```
public List<Empleado> buscarPorNombre(String textoNombre) {
    textoNombre = textoNombre.toLowerCase();
    List<Empleado> encontrados = new ArrayList<>();
    for (Empleado empleado : repositorio)
        if (empleado.getNombre().toLowerCase().contains(textoNombre))
            encontrados.add(empleado);
    return encontrados;
}
```

Filtro por elemento de lista

Otra forma habitual de filtrar para aquellos atributos que tengan un conjunto cerrado de valores, será a través de una lista en la que el usuario pueda seleccionar un valor concreto y muestren solo los elementos que contienen ese valor en el atributo. Vemos un ejemplo de filtro por el género de Empleado, que recordemos que era una enumeración.

Vista:

```
<select id="genero" onChange="changeGenero();">
    <option value="">Todos</option>
    <option th:each="gen : ${T(com.example.myapp.domain.Genero).values()}">
        <th:value="${gen}" th:text="${gen}">
        <th:selected="${gen} == ${generoSeleccionado} ? true : false"></option>
</select>
<script>
    function changeGenero(){
        const select = document.getElementById("genero");
        if (select.value == "") window.location.href = "/";
        else window.location.href = "/findByGenero/" + select.value;
    }
</script>
```

La línea: `th:selected="${gen} == ${generoSeleccionado} ? true : false"` no sería obligatoria, pero es aconsejable para que, cuando muestre los resultados filtrados, en el desplegable aparezca seleccionada la opción que se envió, y no la primera opción u opción por defecto. Como consecuencia, en el *model* habrá que enviarle esa variable '*generoSeleccionado*'.

Controlador:

```
@GetMapping("/findByGenero/{genero}")
public String showFindByGenero(@PathVariable Genero genero, Model model) {
    model.addAttribute("listaEmpleados", empleadoService.buscarPorGenero(genero));
    model.addAttribute("generoSeleccionado", genero);
    return "listView";
}
```

Servicio:

```
public List<Empleado> buscarPorGenero(Genero genero) {
    List<Empleado> encontrados = new ArrayList<>();
    for (Empleado empleado : repositorio)
        if (empleado.getGenero() == genero) encontrados.add(empleado);
    return encontrados;
}
```

Estos dos ejemplos de filtros son independientes, no se suman los filtros por defecto, o se aplica uno u otro. Habría que programar su funcionalidad conjunta.

Atributos no usados en formulario

En algunos casos, por ejemplo, al hacer el alta de un elemento en un CRUD, puede ser que haya atributos que no deseemos que sea el usuario el que introduzca en el formulario, bien por que toman un valor por defecto, bien porque se calculan en un servicio, etc. Un ejemplo, podría ser la fecha de alta de un producto, que se podría tomar del sistema de forma automática.

En estos casos, si la vista con el formulario no tiene ningún `<input>` asociado a un atributo, al hacer el `submit`, ese atributo llegará con valor nulo al controlador. Para solucionar este posible problema tenemos varias opciones.

- Asignar el valor después de recibirlo del `submit` y no antes.
- Si es necesario asignarlo antes, por ejemplo en el constructor, añadir un `<input type="hidden">` en la vista, que hace que permanezca oculto, pero conserve el valor asignado previamente.
- Crear una clase *ad-hoc* solo con los campos de formulario (lo que se llama un DTO), y al recibirlo, mover esos atributos del *dto* al objeto real. Esta sería la mejor opción.

Vamos a ver todo esto en un ejemplo. Supongamos que estamos haciendo el CRUD de una entidad llamada *Producto* que tiene por atributos: *nombre*, *precio* y *fechaRegistro* y que este último atributo se toma de la fecha del sistema. Suponemos también que disponemos de un servicio llamado *ProductoService* con las operaciones sobre el repositorio. Las posibilidades vistas se plasmarían así:

- En el controlador, asignar el valor después de recibirlo del `submit` y no antes.

```
@PostMapping("/nuevo/submit")
public String showNewSubmit(Producto productoForm) {
    productoForm.setFechaRegistro(LocalDate.now());
    productoService.add(productoForm);
    return "productoCreadoView";
}
```

- Si el valor se añadió previamente, añadir un `<input type="hidden">` en la vista del formulario, para no perderlo, entonces, en la clase *Producto*, modificamos el constructor por defecto, para asignarle el valor antes de enviarlo:

```
public Producto (){ this.fechaRegistro=LocalDate.now(); }
```

En la vista:

```
<input type="hidden" th:field="*{fechaRegistro}" />
```

- Crear una clase *ad-hoc* solo con los campos de formulario (lo que se llama un DTO), y al recibirlo, mover esos atributos del *dto* al objeto real. Creamos *ProductoDto*, sin la fecha de registro:

```
public class ProductoDto {
    private String nombre;
    private Double precio;
}
//getters, setters, constructor por defecto
```

A la vista con el formulario le pasamos un *ProductoDto* y no un *Producto*. Al recibir el *dto* llamamos a un método de servicio que asigne los campos del *dto* al producto.

```
@PostMapping("/nuevo/submit")
public String showNewSubmit(ProductoDto productoForm) {
    Producto producto= productoService.buildProductoFromDto(productoForm);
    productoService.add(producto);
    return "redirect:/list";
}
```

El método *buildProductoFromDto* sería algo así:

```
public Producto buildProductoFromDto(ProductoDto productoDto){
    Producto producto = new Producto();
    producto.setNombre (productoDto.getNombre());
    producto.setPrecio (productoDto.getPrecio());
    producto.setFechaRegistro(LocalDate.now());
    return producto;
}
```

Desde Java 14, disponemos de una nueva forma de definir clases sencillas, en una sola línea, con datos inmutables, getters con el mismo nombre que los atributos, `toString()` con todos sus atributos, y `equals()`/`hashCode()` basados también en todos los atributos. Se denominan '**record**' y se definen así:

```
public record ProductoDto (String nombre, Double precio) {}
```

Pueden ser muy útiles para crear DTOs.

Lectura de parámetros de ficheros

Una operación típica de nuestras aplicaciones será obtener determinados valores de variables o constantes de nuestra aplicación desde un archivo de configuración: por ejemplo, el porcentaje de IVA a aplicar en nuestras facturas, importes o fechas globales para toda la aplicación, etc. o incluso los mensajes de texto que se mostrarán dependiendo del idioma seleccionado.

Esos parámetros pueden almacenarse en distintos formatos: xml, json, etc. pero es muy típico el guardarlos en un fichero de tipo *properties*, similar al *application.properties* que ya conocemos.

Este podría ser un ejemplo:

```
tipoIva = 21
maximaOcupacionLocal = 100
comisionVenta = 100
```

La forma de trabajar en Spring con este tipo de archivos es muy sencilla. Estos serían los pasos a seguir:

1.- Crear el archivo con extensión *.properties* y almacenarlo en la carpeta */resources* o en una subcarpeta de esta.

2.- Crear una clase que recoja los parámetros del fichero en variables que luego podremos utilizar. La clase estará en el paquete donde están el resto de clases, típicamente en un subpaquete */config* y deberá cumplir estos requisitos:

- Estar anotada con `@Configuration`. Así se creará automáticamente al inicio de la aplicación.
- Estar anotada con `@PropertySource("classpath:ruta/archivo")`;
- Crear getters y setters (por ejemplo, con anotaciones Lombok: `@Getter` y `@Setter`)
- Tener un atributo para cada variable contenida en el archivo, y cada uno de estos atributos debe estar anotado con `@Value("${nombreVariableEnArchivo}")`

Ejemplo:

```
import org.springframework.beans.factory.annotation.Value;
import org.springframework.context.annotation.Configuration;
import org.springframework.context.annotation.PropertySource;
import lombok.Getter;
import lombok.Setter;

@Configuration
@Getter
@Setter
@PropertySource("classpath:/config/parametrosApp.properties")
public class Parametros {
    @Value("${tipoIva}")
    private Double porcentajeTipoIva;

    @Value("${maximaOcupacionLocal}")
    private Integer maximaOcupacionLocal;

    @Value("${comisionVenta }")
    private Integer importeComisionVenta;
}
```

3.- Luego, simplemente inyectamos la clase en donde sea necesario, generalmente en las clases de servicio:

```
@Autowired  
private Parametros parametros;
```

y usar sus valores normalmente a través de sus getters y setters:

```
salarioEmpleado += parametros.getImporteComisionVenta();
```

Fichero .properties para envío de emails

Podemos usar esta técnica para almacenar parámetros constantes en el envío de correos; por ejemplo, si el destinatario del correo es siempre el mismo (el email del gerente, el Community Manager, etc), podemos meterlo en un fichero de este tipo y leerlo a la hora de enviar el mensaje. También lo podemos hacer con el asunto del mensaje.

Fichero: *resources/emailParams.properties*:

```
email.destination=rdf@fernandowirtz.com  
email.subject="Asunto del mensaje"
```

Clase: *EmailLoadParams.java*:

```
@Configuration  
@Getter  
@Setter  
@PropertySource("classpath:/emailParams.properties")  
public class EmailLoadParams {  
    @Value("${email.destination}")  
    private String emailDestination;  
  
    @Value("${email.subject}")  
    private String emailSubject;  
}
```

Y la llamada sería:

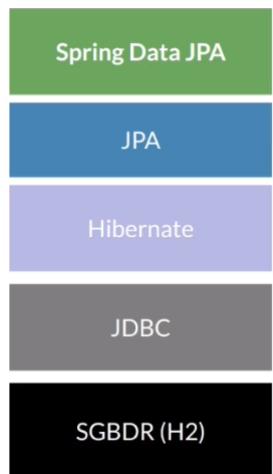
```
@Autowired  
private EmailLoadParams emailLoadParams;  
.  
.  
.  
boolean envioEmail = emailService.sendEmail(emailLoadParams.getEmailDestination(),  
    emailLoadParams.getEmailSubject(), cuerpoMensaje);
```

Tema 7: Acceso a datos

Introducción

Antes de empezar a trabajar con el acceso a datos, debemos definir varios conceptos:

- **Persistencia:** consiste en el almacenamiento de los datos de forma permanente. En las aplicaciones vistas previamente, al finalizar la ejecución, los datos con los que hubiésemos trabajado se perdían. Por persistencia entendemos conservar esos datos en algún soporte que pueda ser accesible posteriormente. Estos soportes tradicionalmente suelen ser ficheros y bases de datos.
- **Repositorio:** como ya vimos en el tema anterior, es una capa adicional de nuestra aplicación que permite gestionar los datos almacenados en alguna colección (generalmente persistente). Habitualmente la capa de servicio hará una solicitud de datos y las clases del repositorio harán el acceso a los datos. *En caso de aplicaciones muy sencillas, se elimina la capa de servicio, y es directamente el controlador el que interactúa con el repositorio.*
- **Bases de datos Relacionales vs. Base de datos NoSQL.** Son los dos modelos actuales de SGBD. El primero es el más clásico y se basa en un almacenamiento en forma de tablas (con filas y columnas) con recursos como la integridad referencial, índices, garantía de no duplicidades, etc. Como inconveniente destaca su dificultad para trabajar con datos no fácilmente "tabulables", es decir cuando cada elemento (fila) puede tener atributos totalmente diferentes (columnas). El modelo NoSQL (not only SQL) permite trabajar con distintos modelos de datos y es mucho más flexible.
- **JDBC:** es el interfaz más básico de Java para conectarnos a cualquier SGBD relacional. Proporciona una API completa para trabajar de forma que sea cual sea el motor de base de datos con el que conectemos, la API siempre será la misma. Simplemente tendremos que obtener el *driver* correspondiente al motor de base de datos que queramos usar, que sí que dependerá totalmente de éste. Trabajando con JDBC debemos conectarnos al SGBD y componer manualmente nosotros las operaciones en SQL. En el caso de consultas, JDBC devolverá una estructura similar a un array bidimensional con filas y columnas y que debemos procesar también manualmente.
- **Hibernate y el Mapeo objeto-relacional (ORM):** Nuestras aplicaciones son orientadas a objetos, pero las bases de datos con las que trabajan están orientadas a tablas/columnas. Son estructuras que no tienen nada que ver entre ellas por lo que se debe hacer su mapeo, haciendo coincidir cada atributo de cada clase con cada campo de una tabla (y viceversa) cada vez que queramos leer o escribir un objeto desde y hacia la base de datos, respectivamente. **Hibernate** es el framework más popular que se encarga de realizar este mapeo de forma transparente para nosotros.
- **JPA:** Define un estándar de forma de trabajo, pero no lo implementa, son los ORM como Hibernate, los que se ajustan a este estándar. JPA define como se relacionan los objetos con el esquema de base de datos, ofrece una API sencilla para realizar las operaciones CRUD y consultas sobre los datos con un lenguaje similar a SQL, pero sobre objetos (JPQL) y otros elementos de optimización. A veces se propone la metáfora: *"sí jpa es el baile, hibernate es el bailarín"*.
- **Spring Data:** es un proyecto "paraguas" que nos permite trabajar con distintas bases de datos con la tecnología que queramos (JDBC, Hibernate, JPA...) sobre bases de datos relacionales o NoSQL, facilita auditoría, configuración sencilla, etc. Nos ofrece clases con multitud de utilidades implementadas, realizando mucho trabajo por nosotros y permitiendo cambiar de soporte (por ejemplo, de un tipo de base de datos a otra) de forma casi transparente para nosotros.



SGBD

Como acabamos de comentar, Spring soporta multitud de sistemas gestores de bases de datos (SGBD), tanto relacionales como no relacionales. En este curso vamos a trabajar principalmente con H2, pero también con MySQL.

H2: Es un gestor de base de datos ligero, que nos permite trabajar con él de forma embebida en nuestro proyecto (mediante Maven como una dependencia más) y muy utilizado en las fases tempranas de desarrollo y también en testing.

Ofrece la posibilidad de trabajar en memoria o bien guardar los datos de forma persistente en disco. Este aspecto lo configuraremos en el fichero *application.properties*. Otra de sus características es su fácil integración con Spring por lo que para nosotros será muy cómodo trabajar con este gestor para evitar instalaciones, conexiones, etc. con gestores de base de datos más potentes.

MySQL: Es un SGBD relacional de código abierto desarrollado bajo licencia dual: Licencia pública general/Licencia comercial y junto con Oracle y SQLServer son las bases más empleadas. En 2009 se creó un fork denominado MariaDB por algunos desarrolladores de MySQL descontentos con el modelo de desarrollo y el hecho de que una misma empresa controle a la vez los productos MySQL y Oracle.

Al contrario de H2, MySQL necesita ser instalado bien en el mismo ordenador que ejecutará nuestra aplicación, bien en otro servidor. En el último capítulo de este manual incluimos un apartado para su instalación pura o mediante un contenedor Docker y también su configuración en proyectos Spring.

Para acceder a los contenidos de nuestras bases de datos MySQL y ejecutar consultas SQL existe una herramienta de escritorio, oficial, de interfaz gráfica llamada MySQL Workbench, pero podríamos usar otras como *phpmyadmin*.

JDBCTemplate

JDBCTemplate es una clase que facilita el acceso a base de datos mediante JDBC. Se encarga de establecer la conexión con la base de datos de forma transparente, ofrece métodos para realizar las consultas SQL de forma sencilla y convierte las filas recibidas a objetos de nuestro dominio. Ejemplo:

```
public Product findById(Long id) {
    try {
        Product product = jdbcTemplate.queryForObject("SELECT * FROM products WHERE id=?", BeanPropertyRowMapper.newInstance(Product.class), id);
        return product;
    } catch (IncorrectResultSizeDataAccessException e) {
        return null;
    }
}

public int save(Product product) {
    return jdbcTemplate.update("INSERT INTO products (name, price) VALUES(?,?)",
        new Object[] { product.getName(), product.getPrice() });
}
```

En los recursos incluidos en este manual se incorpora un ejemplo completo de un CRUD de una entidad sencilla, pero en este manual optaremos por JPA e Hibernate, una opción más sofisticada y que optimiza nuestros tiempos de desarrollo.

Configuración inicial de H2

Para poder trabajar con H2 en nuestros proyectos SpringBoot no necesitamos instalar software adicional, pero deberemos realizar la siguiente configuración:

1.- Agregar H2 al proyecto: Debemos añadir al fichero pom.xml las dependencias **starter-jpa** y la de la base de datos con la que trabajaremos, en nuestro caso emplearemos por ahora **H2**. En el Anexo de MySQL tenemos sus dependencias. Las obtenemos como siempre de

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-data-jpa</artifactId>
</dependency>
<dependency>
    <groupId>com.h2database</groupId>
    <artifactId>h2</artifactId>
    <scope>runtime</scope>
</dependency>
```

Maven permite añadir incluso varios gestores de bases de datos diferentes en el pom.xml y emplear uno en cada entorno, por ejemplo, H2 en desarrollo y testing, y MySQL en producción. Se hace mediante las etiquetas <profile> y <scope>.

2.- Configuración de la base de datos H2: por defecto, Spring Boot configura la aplicación como almacenamiento en memoria (no en disco) y con un usuario administrador llamado 'sa' con contraseña vacía. Debemos añadir estos parámetros en el fichero *application.properties* del proyecto:

```
spring.datasource.url=jdbc:h2:mem:nombreBD
spring.datasource.driverClassName=org.h2.Driver
spring.datasource.username=sa
spring.datasource.password=
spring.jpa.database-platform=org.hibernate.dialect.H2Dialect
```

Desde la consola de H2 podemos cambiar la contraseña del usuario 'sa' mediante la instrucción SQL: `ALTER USER sa SET PASSWORD 'newpass';`

En caso de querer trabajar en disco (y no perder los datos al cerrar el proyecto) cambiaríamos la primera línea: `spring.datasource.url=jdbc:h2:file:./ruta/nombreBD`

para almacenarlo en la ruta especificada. *Generará dos archivos: nombreBD.mv.db y nombreBD.trace.db*. Si empezamos la ruta por punto, creará la/s carpeta/s de la ruta a partir del directorio actual, esto es, la raíz del proyecto.

Por defecto, cada vez que se inicia la aplicación (sea en memoria o fichero) la base de datos será recreada, por lo que, cuando el desarrollo llegue a una fase estable, si no queremos que se lleve ningún cambio sobre el esquema cada vez que arranquemos la aplicación añadiremos la propiedad:

```
spring.jpa.hibernate.ddl-auto = none
```

Otros valores de esta propiedad son: `create` (crear la base de datos de nuevo cada vez), `create-drop` (crear la base de datos al inicializar el servidor y borrarla al pararlo), `validate` (revisar si ha habido cambios en el esquema y provocar una excepción en caso afirmativo) y `update` (actualizar los cambios, pero sin borrar tablas ni columnas).

Otra propiedad interesante, sobre todo en tiempo de desarrollo, es la que indica que aparezcan en el log las consultas SQL que realice la plataforma, para ello añadimos: `spring.jpa.show-sql=true`

Por el contrario, ya en un entorno de producción podemos establecer:

```
spring.h2.console.settings.trace=false
spring.h2.console.settings.web-allow-others=false
```

para evitar que se muestre en el log "trazas" de sus ejecuciones y para deshabilitar el acceso remoto, respectivamente.

3.- Creación del esquema de la base de datos: Como acabamos de comentar, Hibernate por defecto, recrea el esquema de la base de datos en cada ejecución. Otra opción que tenemos, es que, si queremos crear el esquema de forma manual, podemos añadir un script al que llamaríamos *schema.sql* con la definición del esquema (mayormente instrucciones de tipo CREATE TABLE...) en la carpeta *src/main/resources* del proyecto. Debemos tener las siguientes propiedades:

```
spring.jpa.hibernate.ddl-auto=none  
spring.sql.init.mode=always
```

De todas formas, siempre podemos ejecutar ese script, directamente contra el gestor de base de datos. También hay que ser cuidadosos con el orden de creación de tablas en el script, por las dependencias entre unas y otras (claves foráneas).

4.- Inicialización de la base de datos: en el momento que arranque el proyecto podemos hacer la inicialización de las distintas tablas añadiendo datos a las mismas. Lo haríamos mediante un script de sentencias SQL (por ejemplo, INSERTS) al que llamaríamos *data.sql* en la carpeta *src/main/resources* del proyecto. Podemos cambiar este comportamiento por defecto desde el archivo *application.properties* para que no se realice esta carga inicial:

```
spring.sql.init.mode=never
```

Como veremos más adelante, Hibernate creará el esquema de tablas a partir de las entidades (clases) definidas en el proyecto en la inicialización del mismo. Desde la versión 2.5 de Spring, el archivo *data.sql* se ejecuta antes de la inicialización de Hibernate, esto es, antes de crear las tablas, por lo que se podría producir un error al tratar de insertar antes de tener las tablas creadas. Para cambiar este comportamiento y que la inicialización del *data.sql* se produzca después de la inicialización Hibernate en la que crea las tablas, tenemos que añadir una nueva propiedad:

```
spring.jpa.defer-datasource-initialization=true
```

Además de volver a 'always' la propiedad que permite la ejecución de scripts:

```
spring.sql.init.mode=always
```

Así hacemos la carga de las tablas después de la creación del esquema generado por Hibernate. Por otra parte, el uso simultáneo de *data.sql* y *schema.sql* suele ser problemático.

Hay que tener cuidado con la inicialización con datos si trabajamos con persistencia en disco y sin recrear la base de datos en cada ejecución (por ejemplo, cuando la aplicación ya está en producción) ya que podría insertar duplicados. Algo parecido con la creación del esquema, en un entorno de producción debería hacerse una sola vez, y de forma cuidadosa.

5.- Acceso a la consola H2: Este gestor de base de datos tiene una consola en entorno gráfico para ver el contenido de la base de datos y ejecutar sentencias SQL. Por defecto, la consola no está habilitada; para habilitarla debemos añadir la siguiente propiedad al *application.properties*:

```
spring.h2.console.enabled=true
```

Una vez arrancado el proyecto podemos acceder a la consola desde un navegador, en la ruta [/h2-console](#) de la aplicación. Suponiendo que nuestro servidor corre en el puerto 9000, accederíamos mediante: <http://localhost:9000/h2-console> con las mismas credenciales que incluimos en el *application.properties*.

Entidades

Ya describimos en el capítulo anterior lo que era el modelo de dominio de nuestro sistema y lo que eran las entidades del mismo. Comentábamos que, en un ambiente relacional, esas clases serán mapeadas a tablas y los atributos a columnas de las tablas, para lograr su persistencia (en un ambiente NoSQL sería similar, pero hablamos de colecciones en vez de tablas).

Para que ese mapeo sea realizado por Spring Data de forma transparente para nosotros, las clases de nuestro modelo deben cumplir estas características:

- Ser clases POJO (Plain Old Java Object) esto es, una clase simple que no implementa ni hereda ni depende de un framework en especial.
- Estar anotadas con **@Entity** (cuidado con el import: debe ser jakarta.persistence.Entity)
- Tener un atributo (o atributos) que sean clave primaria, anotado con **@Id**. Si queremos que ese identificador sea un valor autonumérico generado por el gestor de base de datos, añadiremos **@GeneratedValue**.

Tener el constructor sin argumentos. Si trabajamos con Lombok simplemente habría que añadirle la anotación: **@NoArgsConstructor**

Ejemplo:

```
@NoArgsConstructor
@Entity
public class Producto {
    @Id
    @GeneratedValue
    private Long idProd
    private String nombre;
    private String descripción;
    private float pvp;
    //otros constructores, getters, setter, y resto de métodos.
```

En este ejemplo, se crearía una tabla con el mismo nombre que la clase y con columnas con el mismo nombre que los atributos, y como hemos comentado el atributo *idProd* sería la PK de la tabla.

En cuanto a los tipos de datos que podemos emplear en los atributos, son válidos los tipos primitivos y sus envoltorios (int, Integer, float, Float, etc.), String, y otros propios de Java como: BigInteger, BigDecimal, util.Date, util.Calendar, sql.Date, sql.Time, sql.Timestamp...

Anotaciones

Además de las que acabamos de mencionar, disponemos de anotaciones adicionales para refinar el comportamiento por defecto del ORM:

@Table (name="nombreTabla") asignaría *nombreTabla* como nombre de la tabla en vez de ponerle el mismo nombre que la clase. Esta anotación iría a continuación de **@Entity**.

@Column (name="nombreColumna") asignaría *nombreColumna* como nombre de la columna en vez de ponerle el mismo nombre que el atributo situado a continuación de la anotación. Esta misma anotación nos permite fijar otras propiedades de la columna y por tanto del atributo (Nullable, Length...). Ejemplo: **@Column (nullable=false)** y una que puede ser muy útil: **@Column(unique=true)** que hará que la columna no contenga duplicados en la tabla, al igual que ocurre con la clave primaria de la tabla. En caso de intentar insertar un duplicado se produce la excepción *DataIntegrityViolationException*..

@UniqueConstraint es similar a **@Column(unique=true)** ya que permite que ciertos atributos no contengan valores duplicados. La diferencia fundamental es que **@UniqueConstraint** se aplica a nivel tabla, no a nivel columna y que podría incluir más de un atributo, lo que significaría que la suma de los dos atributos es la que no puede tener valores repetidos.

```
@Table(uniqueConstraints =
{@UniqueConstraint(columnNames = {"nombre", "fechaNacim"})})
```

En el ejemplo anterior, puede haber nombres duplicados y fechas de nacimiento duplicadas, pero no puede haber dos filas que contengan el mismo nombre y fecha nacimiento a la vez (se lanzaría una *DataIntegrityViolationException*).

`@GeneratedValue` ya mencionada, acompaña a `@Id` y le dice al sistema que ese campo será gestionado de forma automática generando números no repetidos. Se suele utilizar sobre el campo que sea clave primaria y habitualmente es de tipo `Long`. Existen distintas estrategias para generar es número dependiendo del gestor de base de datos subyacente. Por defecto, si no le especificamos ninguna estrategia, es el propio Spring la elige automáticamente según el gestor de base de datos. Si queremos especificarla, con el parámetro `strategy`, podríamos elegir mediante los valores: `auto`, `sequence`, `identity` o `table`, siendo 'auto' el valor por defecto, y que utilizará la estrategia más adecuada para el gestor de base de datos que estemos empleando.

`@GeneratedValue (strategy=GenerationType.IDENTITY)` será una de la que más emplearemos y utilizará los campos de tipo `autoincremental` de gestor de base de datos (ojo si trabajas con Oracle, que no tiene este tipo de datos y no funcionará. Con Oracle es mejor usar la estrategia `sequence`).

En las clases que tienen un `@id` con `@GeneratedValue`, cuando se emplee el constructor que asigna valor a todos los atributos (en Lombok `@AllArgsConstructor`), pondremos un valor por defecto para el atributo `@id`, por ejemplo, cero o null, ya que el valor real que se asignará a ese elemento lo determinará de forma automática el gestor de base de datos. Esto hay que tenerlo en cuenta si queremos crear algún elemento "a mano" desde un `CommandLineRunner`.

Asimismo, disponemos de otras anotaciones ya vistas en el capítulo de formularios:

```
@Min (value=0, message="{producto.id.mayorquecero}")
@NotEmpty
@email
```

Repositories

Spring Data define una interfaz principal llamada `Repository` que permite tomar una entidad con su clave primaria y trabajar sobre ella. Es la interfaz base sobre la que se crea una jerarquía con interfaces más potentes, con métodos muy útiles (que el ORM como Hibernate implementa).

Una de ellas es `CrudRepository` que contiene métodos para las operaciones básicas sobre una tabla referidas habitualmente con el acrónimo CRUD (Create, Read, Update, Delete), un método `count()` para contar filas, etc. Otra interfaz que podríamos usar es `PagingAndSortingRepository` y finalmente `JpaRepository` es la más completa, ofreciéndonos multitud de métodos útiles. Además, `CrudRepository` trabaja más con `Iterable` y `JpaRepository` con `List`, por lo que este último será más cómodo.

Para usarlo, simplemente crearíamos una interfaz con la siguiente estructura:

```
public interface ProductoRepository extends JpaRepository <Producto, Long> {}
```

Y no es necesario añadir ni una sola línea de código más: Spring Data y el ORM se encarga del resto. El nombre que le damos a la interfaz es libre y los dos parámetros que le aportamos a la interfaz genérica `JpaRepository` son la entidad (y por tanto clase y tabla) sobre la que va a trabajar y el tipo de dato de la clave primaria de la entidad (generalmente será de tipo `Long`).

En el ejemplo anterior, estamos definiendo la interfaz `ProductoRepository` sobre la clase `Producto`, que estará anotada con `@Entity` y tendrá un atributo de tipo `Long` anotado con `@Id`.

Interfaces Repository

Vamos a trabajar con `JpaRepository` (y en general con cualquier interfaz derivada de `Repository`) de tres formas diferentes:

- **Métodos definidos en la interfaz.** Son métodos proporcionados por la propia interfaz, los típicos CRUD: añadir, borrar, etc. así como otros para búsqueda, contar, etc. Estos métodos los podremos usar directamente en los servicios o donde deseemos sin crear ni siquiera su firma en el repositorio. Más abajo mostramos la lista de ellos.

- **Métodos derivados:** veremos que, si creamos métodos empleando en su firma unas palabras clave determinadas, Spring Data construirá el método por nosotros. Ejemplo:

```
interface PersonaRepository extends JpaRepository<Persona, Long> {
    List<Persona> findByEmail (String email);
}
```

Spring construirá el método al que, pasándole un email, devolverá una lista con todas las entidades (personas en este caso) que tengan ese email.

- **Métodos query:** Con la anotación `@Query` podremos construir métodos a medida para trabajar con los repositorios y que obtengan listados precisos, joins, etc.

Existen otros métodos de consulta que no trataremos como pueden ser QueryDSL y QueryByExample: **QueryDSL** permite construir consultas a través de un API, mediante predicados, sin necesidad de saber SQL y también válido para Spring Data MongoDB, no solo Spring Data JPA. Ejemplo:

```
Predicate predicate = user.firstName("Pepe").and(user.lastName("López"));
userRepository.findAll(predicate);
```

QueryByExample: se crea un ejemplo de una instancia (Ejemplo) con algunos atributos con valores asignados y que representan los criterios de búsqueda:

```
Empleado filter=new Empleado();
Empleado.setEmail("pepe@gmail.com");
Example<Empleado> example = Example.of(filter);
List<Empleado> result= empleadoRepository.findAll(example);
```

Métodos de Interfaces Repository

La interfaz JpaRepository nos ofrece métodos propios y otros heredados de CrudRepository y PagingAndSortingRepository. La lista completa la puedes consultar en:

<https://docs.spring.io/spring-data/jpa/docs/current/api/org/springframework/data/jpa/repository/JpaRepository.html>

Estos son que más usaremos, siendo T la entidad del repositorio.

Devuelve	Método y Descripción
List<T>	findAll()
List<T>	findAll(Sort sort)
List<T>	findAllById(Iterable<ID> ids)
void	flush() Flushes all pending changes to the database.
T	getReferenceById(ID id) Returns a reference to the entity with the given identifier. <i>(sustituye a getByid y getOne, que están deprecated)</i>
<S extends T> List<S>	saveAll(Iterable<S> entities)
<S extends T> List<S>	saveAllAndFlush(Iterable<S> entities) Saves all entities and flushes changes instantly.
<S extends T> S	saveAndFlush(S entity) Saves an entity and flushes changes instantly.

Heredados de CRURepository:

Devuelve	Método y Descripción
long	count() Returns the number of entities available.
void	delete(T entity) Deletes a given entity.
void	deleteAll() Deletes all entities managed by the repository.
void	deleteByid(ID id) Deletes the entity with the given id.
boolean	existsByid(ID id) Returns whether an entity with the given id exists.
Iterable<T>	findAll() Returns all instances of the type.

<code>Optional<T></code>	<code>findById(ID id)</code> Retrieves an entity by its id.
<code><S extends T></code>	<code>save(S entity)</code> Saves a given entity.
<code><S extends T></code>	<code>saveAll(Iterable<S> entities)</code> Saves all given entities.

Métodos derivados

Como ya comentamos, estos métodos definidos sobre un repositorio JPA para obtener datos del mismo son construidos automáticamente por el framework, pero debemos seguir una serie de reglas a la hora de definir su nombre. Si lo hacemos correctamente podremos usarlos como los métodos vistos en el apartado anterior.

Los métodos se incluirán en la definición de la interfaz que extiende JPARepository (o CRUDRepository, PageAndSortingRepository, etc.). Las reglas que tenemos que seguir son las siguientes:

1.- El nombre del método debe empezar por: `findById` aunque también es válido `countBy`, que contaría instancias devolviendo un Long, y otros como: `getBy`, `queryBy` o `readBy`.

2.- A continuación, pondremos el nombre de atributo que usaremos para filtrar la obtención de resultados. Pueden ser varios unidos por "And" o "Or". Por convención, debemos emplear notación "camelcase", es decir, el principio de cada palabra (incluyendo los nombres de atributos) deberá empezar por mayúscula y el resto en minúscula. Ejemplos:

```
List<Empleado> findByNombre (String nombre);
List<Empleado> findByNombreAndEmail (String nombre, String email);
```

Podemos añadirle `Is` o `Equals` para mejor legibilidad, aunque no es necesario:

```
List<Empleado> findByNombreEquals (String nombre);
```

3.- Podemos restringir el número de resultados devueltos insertando la partícula "First" o "Top" y la cantidad de resultados que queremos obtener entre "find" y "By". Ejemplo:

```
List<Empleado> findTop3ByNombre (String nombre);
```

4.- Podemos usar otras partículas para componer las consultas como: `And`, `Or`, `Is`, `Equals`, `Between`, `LessThan`, `LessThanEqual`, `GreaterThan`, `GreaterThanOrEqual`, `After`, `Before`, `IsNull`, `IsNotNull`, `NotNull`, `Like`, `NotLike`, `StartingWith`, `EndingWith`, `Containing`, `OrderBy`, `Not`, `In`, `NotIn`, `True`, `False`, `IgnoreCase`. Ejemplos:

```
List<Empleado> findByEdadBetween (int edadInicial, int edadFinal);
List<Empleado> findByNombreContainingIgnoreCase(String nombre);
List<Empleado> findByEmailIsNotNull();
```

Podemos complicarlo un poco...

```
List<Empleado> findByNombreStartingWithAndIdLessThan (String nombre, Long id);
List<Empleado> findByEmailInAndFechaNacimientoLessThan(List<String> emails,
                                                       Date birth);
```

5.- Podemos ordenar los resultados mediante la partícula final "OrderBy" más un atributo y luego "Asc", o "Desc" para ordenar ascendentemente o descendentemente los resultados. "Asc" es opcional, solo por mejorar la legibilidad, ya que la ordenación por defecto es ascendente.

```
List<Empleado> findByNombreOrderByAsc (String nombre);
List<Empleado> findByNombreOrderByDesc (String nombre);
```

6.- Combinando la ordenación vista en el punto anterior con la reducción del número de resultados con `Top` o `First` podemos obtener máximos o mínimos de forma sencilla. Los siguientes ejemplos obtendrían el empleado con el salario más alto de la base de datos y el empleado con el salario más alto del departamento pasado como parámetro, respectivamente.

```
Empleado findTopOrderBySalarioDesc ();
Empleado findTopByDepartamentoOrderBySalarioDesc (Departamento departamento);
```

Métodos @Query

JPQL y @Query

Cuando la consulta que necesitamos no se puede resolver por ninguno de los dos sistemas anteriores (los métodos de la interfaz o los métodos derivados de nombre) JPA nos ofrece una forma adicional de componer consultas a medida con lenguaje con la anotación @Query y el lenguaje JPQL.

Nos llevaría mucho tiempo profundizar en el lenguaje JPQL, pero podemos resumir que es una mezcla entre SQL y orientación a objetos. Así pues, una consulta típica de SQL como podría ser:

```
select * from Empleado e where e.id = 1
```

En JPQL sería:

```
select e from Empleado e where e.id = 1
```

Al llamar a una consulta JPQL el resultado se guardará en un tipo primitivo o wrapper class, en una instancia del objeto subyacente o en una lista de objetos, dependiendo que lo que hayamos especificado en la parte *select*.

Este sería un ejemplo que obtendría el empleado con el número de id más alto.

```
@Query("select e from Empleado e where e.id=(select max(e2.id) from Empleado e2)")  
Empleado queryMaxIdEmpleado();
```

En la anotación `@Query` incluimos la consulta JPQL y a continuación la firma del método que invocará esa consulta. **Estas consultas están definidas en el repositorio y se invocarán desde una clase de tipo `@Service` de nuestra aplicación que tenga inyectado dicho repositorio.**

Otro ejemplo podría ser el siguiente que obtiene la suma de todos los salarios de todos los empleados:

```
@Query("select sum(e.salario) from Empleado e")  
Optional <Double> querySumSalarios();
```

La operación `sum()` puede devolver `null` si la tabla de empleados está vacía, de ahí que la respuesta sea un `Optional<Double>` en vez de `Double`, así nos aseguramos no tener excepciones por valores nulos.

```
public Double obtenerSumaSalario(){  
    return repositorio.querySumSalarios().orElse(0d);  
}
```

Parámetros en @Query

Obviamente la consulta puede recibir parámetros. Tenemos varias formas de hacerlo, una es empleando el símbolo la interrogación "?" y un número que indica el orden dentro de los parámetros pasados, empezando en 1. Ejemplo:

```
@Query("select e from Empleado e where e.nombre=?1 and e.email=?2")  
Empleado obtenerEmpleadoPorNombreYEmail (String nombre, String email);
```

La otra forma de pasar un parámetro, y es la recomendada, es por nombre y no por posición. Usaríamos la anotación `@Param`. El ejemplo quedaría así:

```
@Query("select e from Empleado e where e.nombre=:nombre and e.email=:email")  
Empleado obtenerEmpleadoPorNombreYEmail (@Param("nombre") String nombre,  
                                         @Param("email") String email);
```

Spring Data también es compatible con SQL nativo, añadiendo el parámetro `nativeQuery` a la consulta. Ejemplo:

```
@Query(nativeQuery = true, value = "select count(1) from Empleado")  
long obtenerTotalFilas();
```

Actualizaciones con @Query

@Query también admite operaciones de actualización y eliminación con JPQL (no inserciones) añadiendo la anotación **`@Modifying`**.

```

@Modifying
@Query("update Empleado e set e.email = :email where e.id = :id")
int updateEmailById(@Param("id") Integer id, @Param("email") String email);

```

Este tipo de operaciones requiere que la llamada a este método (que se hará desde un servicio) esté anotada con **@Transactional**. Más adelante hablaremos de esta anotación. Ejemplo:

```

@Service
public class EmpleadoServiceImpl implements EmpleadoService {
    @Autowired
    private EmpleadoRepository emplRepository;
    @Transactional
    public int actualizarEmail(Integer id, String email) {
        int i = emplRepository.updateEmailById(id, email);
        return i;
    }
}

```

Joins en @Query

Podemos insertar joins entre diferentes entidades, pero deben tener una asociación definida previamente (en apartados posteriores veremos estas asociaciones: uno a muchos, muchos a muchos, etc.). Podremos emplear INNER JOIN y OUTER JOIN.

Proyecto con repositorio persistente

Partiendo del ejemplo del capítulo anterior, en el que hacíamos un CRUD de una entidad Empleado sobre un repositorio en memoria (mediante una colección de tipo List) vamos a hacerla persistente sobre una base de datos H2 e incluir todos los conceptos que acabamos de ver. Podemos hacer una copia de ese proyecto y estos serían los pasos a seguir.

1.- Actualizar el *pom.xml*/añadiendo las dependencias de JPA y H2.

```

<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-data-jpa</artifactId>
</dependency>
<dependency>
    <groupId>com.h2database</groupId>
    <artifactId>h2</artifactId>
    <scope>runtime</scope>
</dependency>

```

2.- Actualizar el fichero *application.properties* tal y como hemos visto en el apartado de configuración, en principio con la base de datos en memoria y, cuando todo funcione correctamente, cambiándolo a disco. También añadimos el resto de parámetros: usuario y contraseña, habilitar la consola, etc.

```

spring.datasource.url=jdbc:h2:mem:mydb
#para crearla en disco
#spring.datasource.url=jdbc:h2:file:./ruta/nombreBD

#para crearla cada vez (con nuevos cambios)
spring.jpa.hibernate.ddl-auto = create
#para no hacer cambios, solo validar que la estructura BD igual a las clases
#spring.jpa.hibernate.ddl-auto = validate
spring.datasource.driverClassName=org.h2.Driver
spring.datasource.username=sa
spring.datasource.password=
spring.jpa.database-platform=org.hibernate.dialect.H2Dialect
spring.jpa.show-sql=true
spring.h2.console.enabled=true

```

No puede haber espacios en blanco al final de cada línea, sino se produce error y no deja arrancar la aplicación

3.- Repasar la clase Empleado para comprobar si tiene getters, setters y constructores (con o sin Lombok). Añadirle las anotaciones `@Entity`, `@Id` y opcionalmente otras anotaciones como `@GeneratedValue`.

```

@Data
@AllArgsConstructor
@NoArgsConstructor
@EqualsAndHashCode(of = "id")

@Entity
public class Empleado {
    @Id
    @GeneratedValue (strategy=GenerationType.IDENTITY)
    private Long id;

    @NotEmpty
    private String nombre;
    @Email
    private String email;
    private Double salario;
    private boolean enActivo;
    private Genero genero;
}

```

4.- Añadir un nuevo archivo `EmpleadoRepository.java` con esta interfaz que extiende `JpaRepository` sobre `Empleado`.

```
public interface EmpleadoRepository extends JpaRepository<Empleado, Long> {}
```

5.- Crear un servicio al que llamaremos por ejemplo `ServiceEmpleadoImplBD` que será similar a `ServiceEmpleadoImplMem` del proyecto del tema anterior, y que también implemente la interfaz `ServiceEmpleado`. Esta nueva clase tendrá inyectado el repositorio `EmpleadoRepository`. Los métodos se denominan igual que en otro proyecto, pero en vez de actuar sobre aquella List en memoria, ahora se invocan a los métodos del repositorio (son los métodos que tenemos automáticamente al extender `JpaRepository`).

```

@Service
public class EmpleadoServiceImplBD implements EmpleadoService {
    @Autowired
    private EmpleadoRepository repositorio;
    public List<Empleado> obtenerTodos() {
        return repositorio.findAll ();
        //si queremos que el resultado esté ordenado por un determinado atributo
        //podemos usar la versión de findAll al que se le incorpora Sort. Ejemplos:
        //    return repositorio.findAll(Sort.by(Sort.Direction.ASC, "email"));
        //    return repositorio.findAll(Sort.by(Sort.Direction.DESC, "salario"));
    }

    public Empleado obtenerPorId (long id) throws RuntimeException {
        return repositorio.findById(id).orElseThrow(
            () -> new RuntimeException("Empleado no encontrado"));
        // findById de JpaRepository devuelve un Optional. Para simplificar,
        // y que el servicio siga devolviendo Empleado y no Optional<Empleado>
        // hacemos que si no lo encuentra lance una excepción. La otra opción sería
        // que devolviese null:
        //    return repositorio.findById(id).orElse(null);
    }

    public Empleado añadir (Empleado empleado) {
        // añadiríamos lógica de negocio. P.ej: guardar si salario > 18000
        return repositorio.save (empleado);
    }
}

```

```

public Empleado editar (Empleado empleado) throws RuntimeException {
    obtenerPorId(empleado.getId()); // Lanza excepción si no existe
    return repositorio.save (empleado);
}

public void borrar(Long id) throws RuntimeException {
    obtenerPorId(id); // opcional:Lanza excepción si no existe
    repositorio.deleteById (id);
}
}

```

En los ejemplos sencillos vamos a ver que la capa de servicio apenas hace prácticamente nada, simplemente recibe los parámetros del controlador e invoca a los métodos del repositorio. Podríamos pensar en eliminar esta capa, aunque por metodología, es mejor mantenerla. **En proyectos reales, más complejos, los métodos de los servicios sí que incorporarán más lógica y probablemente invoquen a métodos de distintos repositorios.** Por ejemplo, en un método de un servicio que realice el alta de cliente, a lo mejor hay que trabajar con el repositorio de datos de cliente, el repositorio de datos de facturación, hacer validaciones, enviar un email al cliente, etc. Todo ello podría ser un mismo método en la capa de servicio.

6.- El controlador será prácticamente igual al del proyecto del tema anterior. La división en capas de la aplicación hace que todos los mappings no se vean afectados por los cambios realizados (es una de las ventajas de la división en capas). El único problema que tenemos es que el controlador tiene autoinyectada la interfaz *ServiceEmpleado* pero tenemos dos clases que la implementan: que tomamos del proyecto del tema anterior: *ServiceEmpleadoImplMem* y la nueva: *ServiceEmpleadoImplBD*. Esto representa un conflicto para Spring, para resolverlo añadiremos a *ServiceEmpleadoImplBD* la anotación **@Primary** para indicarle que, por defecto, use esta implementación.

```

@Service
@Primary
public class EmpleadoServiceImplBD implements EmpleadoService { . . .

```

La otra opción sería directamente eliminar la implementación en memoria del tema anterior.

7.- Las vistas son exactamente igual salvo que el “id” de Empleado (al haberle añadido *@GeneratedValue*), tanto en el formulario de alta como en el de edición, ya no tiene sentido que sea editable pues es Hibernate quien lo gestiona internamente. En el formulario de alta podemos eliminar este campo del formulario ya que en ese momento aún no tiene ningún valor, será al insertarlo en el repositorio cuando tomará un valor autonumérico. En el formulario de edición le añadiremos el atributo “**readonly**” o bien hacerlo “**hidden**”. **Ojo: no usar el atributo “disabled” que no funciona bien en los formularios de edición.**

```

<label>Id:<input type="hidden" id="id" th:field="*{id}"/> </label><br/> o bien:
<label>Id:<input type="text" readonly id="id" th:field="*{id}"/> </label>

```

8.- En el proyecto anterior, mediante un *CommandLineRunner* añadimos un par de empleados, ahora con *id* igual a null, ya que al emplear *@GeneratedValue* en la clase, el *id* se asignará de forma automática:

```

@Bean
CommandLineRunner initData(EmpleadoService empleadoService) {
    return args -> {empleadoService.adadir(new Empleado(null, "pepe", "pepe@gmail.com",
        1000d, true, Genero.MASCULINO));
        empleadoService.adadir(new Empleado(null, "ana", "ana@gmail.com",
        2000d, true, Genero.FEMENINO));
    };
}

```

9.- Podemos añadir a nuestra interfaz algún método derivado de nombre, por ejemplo, obtener los empleados con un salario mayor o igual a un valor pasado como parámetro, ordenados por salario ascendenteamente.

```
public interface EmpleadoRepository extends JpaRepository<Empleado, Long> {
    List<Empleado> findBySalarioGreaterThanOrEqualToSalario (double salario);
}
```

10.- Y por último añadimos a la misma interfaz un método a medida hecho con @Query, por ejemplo, que devuelva los empleados que cobran un salario mayor que el salario medio.

```
public interface EmpleadoRepository extends JpaRepository<Empleado, Long> {
    List<Empleado> findBySalarioGreaterThanOrEqualToSalario (double salario);

    @Query("select e from Empleado e " +
           "where e.salario > (select avg (e2.salario) from Empleado e2)")
    List <Empleado> queryBySalarioOverAverage();
}
```

11. Tenemos que incorporar estos dos nuevos métodos del repositorio a la aplicación, esto es: añadirlos al servicio (Interfaz) EmpleadoService: podemos llamarlos igual:

```
public interface EmpleadoService {
    Empleado añadir(Empleado empleado);
    List<Empleado> obtenerTodos();
    Empleado obtenerPorId(long id);
    Empleado editar(Empleado empleado);
    void borrar(Long id);
    List<Empleado> obtenerEmpleadosSalarioMayor (double salario);
    List<Empleado> obtenerEmpleadoSalarioMayorMedia();
}
```

Y a la implementación del Servicio. En realidad, el servicio solo invoca al método de repositorio, es un caso muy sencillo ya que no incorpora lógica de negocio alguna:

```
@Service
public class EmpleadoServiceImplBD implements EmpleadoService {
    @Autowired
    private EmpleadoRepository repositorio;
    ...
    public List<Empleado> obtenerEmpleadosSalarioMayor (double salario){
        return repositorio.findBySalarioGreaterThanOrEqualToSalario(salario);
    }
    public List<Empleado> obtenerEmpleadoSalarioMayorMedia() {
        return repositorio.queryBySalarioOverAverage ();
    }
}
```

12.- Finalmente modificamos el controlador y las vistas para incorporar estos listados. Por una parte: en el controlador incorporamos los mapping y hacemos que devuelvan los dos la misma vista (es una tabla de empleados) la diferencia será solo el título de la vista.

```
@GetMapping("/listado1/{salario}")
public String showListado1(@PathVariable Double salario, Model model) {
    List<Empleado> empleados = empleadoService.obtenerEmpleadosSalarioMayor (salario);
    model.addAttribute("tituloListado", "Empleados salario mayor que " +
                           salario.toString());
    model.addAttribute("listaEmpleados", empleados);
    return "listadosView";
}
@GetMapping("/listado2")
public String showListado2(Model model) {
    List<Empleado> empleados = empleadoService.obtenerEmpleadoSalarioMayorMedia();
    model.addAttribute("tituloListado", "Empleados salario mayor que la media");
    model.addAttribute("listaEmpleados", empleados);
    return "listadosView";
}
```

Incorporamos esa nueva vista `/listadosView` con el siguiente `<body>`:

```

<h1 th:text="${tituloListado}">Título listado</h1>
<table>
  <thead>
    <th>ID</th><th>Nombre</th><th>Salario</th>
  </thead>
  <tbody>
    <tr th:each="empleado : ${listaEmpleados}">
      <td th:text="${empleado.id}">Id</td>
      <td><a th:href="@{/id}(id=${empleado.id})" th:text="${empleado.nombre}">nombre</a></td>
      <td th:text="${empleado.salario}">salario</td>
    </tr>
  </tbody>
</table>
<a th:href="@{/}">Inicio</a><br/>

```

Y para que el usuario acceda a esos listados, en la vista con la tabla de empleados inicial, añadiremos dos enlaces que invoquen a sendos listados. El primer listado necesita el valor de corte del salario a mostrar. Podríamos hacer un formulario con una caja de texto para obtener ese importe, pero para hacerlo más simple, hacemos una función JavaScript sencilla:

```

Salario filtro:<input type="text" id="salario"/>
<input type="button" onclick="consultarSalario()" value="Salario superior" />
<script>
  function consultarSalario(){
    window.location.href = "/listado1/" + document.getElementById("salario").value;
  }
</script>
<a th:href="@{/listado2}">Listado 2 (salario > media)</a><br/>

```

13.- En la vista con el formulario para añadir un nuevo empleado, el campo correspondiente al id no será `type="text"`, será `type="hidden"` ya que es el propio sistema el que lo gestiona al ser autogenerado. En la vista de edición de empleado podemos mantenerlo de tipo "text", pero añadiéndole el atributo `readonly` para que no se pueda modificar.

14.- Un último paso sería añadirle una imagen a cada empleado. Para ello hay que seguir los pasos mostrados en el capítulo 5 en cuanto a la subida y almacenamiento de archivos, pero adicionalmente dispones del video siguiente que explica el proceso paso a paso: https://youtu.be/_NjSWG8bs5I

Mapeo de asociaciones

Las entidades de nuestro modelo (clases) no están aisladas, sino que se relacionan entre ellas, así pues, si tenemos en nuestro modelo de dominio una entidad "Libro" y una entidad "Autor" lo lógico es que exista una relación entre ellos. Los diagramas de clase UML vistos en el curso anterior son una forma de representación habitual de estas relaciones.

Un aspecto importante de estas asociaciones entre entidades es la cardinalidad o multiplicidad, es decir cuántos elementos de una entidad se pueden relacionar con los de la otra. Así hablamos de relaciones: "uno a uno", "uno a muchos" (y "muchos a uno") y "muchos a muchos". En el ejemplo que hemos puesto hablaríamos de una relación "muchos a muchos" ya que un libro puede estar escrito por uno o varios autores, y un autor puede escribir cero o muchos libros. Si tuviésemos una entidad "Ejemplar" podríamos establecer una relación "uno a muchos" entre Libro y Ejemplar, ya que un libro tendrá muchos ejemplares, pero un ejemplar solo puede pertenecer a un solo Libro.

JPA nos permite representar las asociaciones entre clases mediante una serie de anotaciones en las entidades. Dependiendo de su multiplicidad. Son las siguientes:

- @ManyToOne
- @OneToMany
- @ManyToMany
- @OneToOne

Estas relaciones se establecerán generando claves foráneas o tablas adicionales y realizarán los *join* necesarios cuando vinculemos instancias de las clases involucradas en la relación; todo de forma transparente para nosotros. ¡Hurra!

@ManyToOne

Asociación muy frecuente: "muchos a uno". El siguiente ejemplo muestra como un Empleado tiene una asociación "muchos a uno" con la entidad Departamento, ya que en un departamento puede haber muchos empleados, pero un empleado solo puede pertenecer a un departamento.

Es una de las relaciones que más nos vamos a encontrar y para resolverla añadiremos la anotación **@ManyToOne** en la entidad "muchos" y esa anotación hará referencia al elemento "uno" de la relación.

```
@Entity
public class Empleado {
    @Id
    @GeneratedValue (strategy=GenerationType.IDENTITY)
    private Long id;
    @ManyToOne
    private Departamento departamento; //Departamento debe ser también @Entity
    //resto de atributos, constructores, getters, setter, y resto de métodos.
    //Si tenemos constructores no Lombok deben incluir ahora el departamento
```

La clase Departamento podría ser así:

```
@Data
@AllArgsConstructor
@NoArgsConstructor

@Entity
public class Departamento {
    @Id
    @GeneratedValue (strategy=GenerationType.IDENTITY)
    private Long id;

    @NotEmpty
    private String nombre;
}
```

Si consultamos la consola H2 después de iniciar la aplicación podemos ver que en la tabla *Empleado* añade una nueva columna llamada *Departamento_id* y que el nombre de la restricción es generado con caracteres al azar.

The screenshot shows the H2 Database Console interface. The left sidebar displays the database schema with tables 'DEPARTAMENTO' and 'EMPLEADO'. The 'DEPARTAMENTO_ID' column in the 'EMPLEADO' table is circled in red. The right pane shows the results of a SQL query to select constraints from the 'REFERENTIAL_CONSTRAINTS' table. A specific constraint is highlighted with a red circle, showing the following details:

CONSTRAINT_CATALOG	CONSTRAINT_SCHEMA	CONSTRAINT_NAME	UNIQUE_CONSTRAINT
EMPLEADOBD	PUBLIC	FKHDJJHOHPYJSFTA5G6P8B8E00	EMPLEADOBD

Opcionalmente, podemos añadir las anotaciones `@JoinColumn`, que nos permite indicar el nombre de la columna que hará las funciones de clave externa, así como `@ForeignKey`, con la que podemos indicar el nombre de la restricción que se creará a nivel de base de datos (muy útil para depurar errores).

```

@Entity
public class Empleado {
    @Id
    @GeneratedValue (strategy=GenerationType.IDENTITY)
    private Long id;

    @ManyToOne
    @JoinColumn(name = "DEPTO_ID", foreignKey = @ForeignKey(name="DEPTO_ID_FK"))
    private Departamento departamento;

    // resto de atributos
    // constructores, getters, setter, y resto de métodos.
    // Si tenemos constructores no Lombok deben incluir ahora el departamento

```

Los pasos a seguir para incorporar esta nueva entidad y relación al proyecto serían:

- 1) Crear la clase Departamento según acabamos de ver.
- 2) Modificar la clase Empleado añadiendo la nueva relación.
- 3) Crear un JpaRepository (u otra interfaz Repository como CrudRepository) para Departamento. Podemos añadirle métodos derivados por nombre, métodos @Query, etc. En este caso le vamos a añadir el método derivado para obtener un departamento a partir de su nombre aunque, en principio, no es necesario:

```

public interface DepartamentoRepository extends JpaRepository<Departamento, Long> {
    Departamento findByNombre(String nombre);
}

```

En los métodos derivados por nombre, cuando hay relaciones entre entidades, podemos definir métodos que hagan referencia a las entidades relacionadas, siguiendo el ejemplo de los apuntes, podríamos, en el repositorio de Empleado, incluir el método:

```
List<Empleado> findByDepartamento(Departamento departamento);
```

Pero también podríamos hacer referencia a los atributos de la entidad relacionada, empleando notación *camelcase*. Por ejemplo, en el repositorio anterior podríamos haber incluido:

```
List<Empleado> findByDepartamentoId (Long idDept);
List<Empleado> findByDepartamentoNombre (String nomDept);
```

- 4) Crear una interfaz de servicio para las operaciones CRUD de Departamento.

```

public interface DepartamentoService {
    Departamento añadir(Departamento departamento);
    List<Departamento> obtenerTodos();
    Departamento obtenerPorId(long id);
    Departamento editar(Departamento departamento);
    void borrar(Long id);
    Departamento obtenerPorNombre(String nombre);
}

```

Y la clase que lo implementa:

```

@Service
public class DepartamentoServiceImplBD implements DepartamentoService {
    @Autowired
    private DepartamentoRepository departamentoRepository;

    public Departamento añadir(Departamento departamento) {
        return departamentoRepository.save(departamento);
    }
}

```

```

public List<Departamento> obtenerTodos() {
    return departamentoRepository.findAll();
}

public Departamento obtenerPorId(long id) {
    return departamentoRepository.findById(id).orElse(null);
}

public Departamento editar(Departamento departamento) {
    return departamentoRepository.save(departamento);
}

public void borrar(Long id) {
    departamentoRepository.deleteById(id);
    //también podemos usar delete, pasándole un departamento, no un id
}

public Departamento obtenerPorNombre(String nombre) {
    return departamentoRepository.findByNombre(nombre);
}
}

```

- 5) Crear un controlador para acceder a los métodos del servicio del punto anterior. Podríamos asignarle un `@RequestMapping ("~/depto")` a nivel clase para que se aplique a todos sus métodos (será análogo al de Empleado). No vamos a generar excepciones para hacerlo más simple.

```

@Controller
@RequestMapping("/depto")
public class DepartamentoController {

    @Autowired
    private DepartamentoService departamentoService;

    @GetMapping({ "/", "/list" })
    public String showList(Model model) {
        model.addAttribute("listaDepartamentos", departamentoService.obtenerTodos());
        return "departamento/listView";
    }

    @GetMapping("/new")
    public String showNew(Model model) {
        model.addAttribute("departamentoForm", new Departamento());
        return "departamento/newFormView";
    }

    @PostMapping("/new/submit")
    public String showNewSubmit( @Valid Departamento departamentoForm,
                               BindingResult bindingResult) {
        if (bindingResult.hasErrors()) return "redirect:/depto/new";
        departamentoService.anadir(departamentoForm);
        return "redirect:/depto/list";
    }

    @GetMapping("/edit/{id}")
    public String showEditForm(@PathVariable long id, Model model) {
        Departamento departamento = departamentoService.obtenerPorId(id);
        if (departamento != null) {
            model.addAttribute("departamentoForm", departamento);
            return "departamento/editFormView";
        } else return "redirect:/depto/list";
    }

    @PostMapping("/edit/submit")
    public String showEditSubmit( @Valid Departamento departamentoForm,
                               BindingResult bindingResult) {
        if (!bindingResult.hasErrors())
            departamentoService.editar(departamentoForm);
        return "redirect:/depto/list";
    }
}

```

```

    @GetMapping("/delete/{id}")
    public String showDelete(@PathVariable long id) {
        departamentoService.borrar(id);
        return "redirect:/depto/list";
    }
}

```

- 6) Hay que inyectar el servicio de gestión de departamentos en el controlador de empleado ya que cuando demos de alta un nuevo empleado, habrá que pasarle la lista de departamentos a la vista para que elija uno (y también para la modificación de empleados).

```

@Controller
public class EmpleadoController {
    @Autowired
    private EmpleadoService empleadoService;
    @Autowired
    private DepartamentoService departamentoService;

    @GetMapping("/new")
    public String showNew(Model model) {
        model.addAttribute("empleadoForm", new Empleado());
        model.addAttribute("listaDepartamentos", departamentoService.obtenerTodos());
        return "empleado/newFormView";
    }

    @GetMapping("/editar/{id}")
    public String showEditForm(@PathVariable long id, Model model) {
        try {
            Empleado empleado = empleadoService.obtenerPorId(id);
            model.addAttribute("empleadoForm", empleado);
            model.addAttribute("listaDepartamentos", departamentoService.obtenerTodos());
            return "empleado/editFormView";
        } catch (RuntimeException e) { . . . }
        return "redirect:/";
    }
}

```

- 7) Crear nuevas vistas relacionadas con el controlador anterior y modificar las de empleado para que traten el departamento. Podemos estructurar las vistas en la carpeta *templates* haciendo dos subcarpetas, una para empleado y otra para departamento. La vista general de departamentos podría contener:

```

<h1>Listado de Departamentos</h1>
<table>
    <thead>
        <tr><th>ID</th><th>Nombre</th><th>Editar</th><th>Borrar</th></tr>
    </thead>
    <tbody>
        <tr th:each="departamento : ${listaDepartamentos}">
            <td th:text="${departamento.id}">Id</td>
            <td th:text="${departamento.nombre}">nombre</td>
            <td><a th:href="@{/depto/edit/{id}(id=${departamento.id})}">
                Editar</a></td>
            <td><a th:href="@{/depto/delete/{id}(id=${departamento.id})}">
                Borrar</a></td>
        </tr>
    </tbody>
</table>
<a th:href="@{/depto/new}">Nuevo Departamento</a><br/>

```

- 8) En la vista de nuevo empleado y de edición de empleado, el id de departamento se asignará a '*departamento*', que es como se llama el atributo y no *idDepartamento*.

```
Departamento:<br/>
<select name="list1" th:field="*{departamento}">
    <option value="0">select option</option>      <!--opcion por defecto-->
    <option th:each="departamento:${listaDepartamentos}"
            th:value="${departamento.id}" th:text="${departamento.nombre}">
    </option>
</select>
```

- 9) Una mejora interesante sería incluir en la vista de la lista de empleados un desplegable con los departamentos, que permita seleccionar uno y mostrar solo sus empleados. Para ello debemos añadir en la vista de la lista de empleados un *<select>*:

```
<select id="select" onChange="changeDepartamento();">
    <option value="0">Todos</option>
    <option th:each="dep : ${listaDepartamentos}"
            th:value="${dep.id}" th:text="${dep.nombre}"
            th:selected="${dep.id} == ${deptoSeleccionado} ? true : false">
    </option>
</select>
. . .
<script>
    function changeDepartamento(){
        const select = document.getElementById("select");
        if (select.value == 0) window.location.href = "/";
        else window.location.href = "/porDepto/" + select.value;
    }
</script>
```

Y en el controlador debemos adaptarnos a esos cambios en la vista, es decir, debemos pasarle la lista de departamentos y el departamento seleccionado (que inicialmente será cero o nulo):

```
@GetMapping({ "/", "/list" })
public String showList(Model model) {
    model.addAttribute("listaEmpleados", empleadoService.obtenerTodos());
    model.addAttribute("listaDepartamentos", departamentoService.obtenerTodos());
    model.addAttribute("deptoSeleccionado", 0); // new Departamento(0L, "Todos"));
    . . .
```

Y por otra parte crear el nuevo mapping */porDepto/{idDepto}* donde si le devolveremos a la vista el departamento seleccionado.

```
@GetMapping("/porDepto/{idDepto}")
public String showbyDepto(@PathVariable Long idDepto, Model model) {
    model.addAttribute("listaEmpleados",
                      empleadoService.obtenerPorDepartamento(idDepto));
    model.addAttribute("listaDepartamentos", departamentoService.obtenerTodos());
    model.addAttribute("deptoSeleccionado", idDepto);
    model.addAttribute("findForm", new Empleado());
    if (txtMsg != null) {
        model.addAttribute("msg", txtMsg);
        txtMsg = null;
    }
    return "empleado/listView";
}
```

- 10) En la vista inicial de la aplicación deberíamos incluir un enlace a una vista que mostrarse los departamentos y permitiese su gestión, análoga a la de Empleado.

```
<a th:href="@{/depto/}"/>Gestión de departamentos</a><br/>
```

11) En el *CommandLineRunner* podemos añadir departamentos de prueba antes de añadir los empleados.

```
@Bean
CommandLineRunner initData(EmpleadoService empleadoService,
                           DepartamentoService departamentoService) {
    return args -> {
        Departamento depInf =
            departamentoService.añadir(new Departamento(0L, "Informática"));
        Departamento depRRHH = departamentoService.añadir(new Departamento(0L, "RRHH"));
        empleadoService.añadir(new Empleado
            (null, "pepe", "pepe@gmail.com", 28000d, true, Genero.MASCULINO, depInf));
    };
}
```

Con todo esto, deberíamos poder ejecutar la aplicación perfectamente, pero tendremos un problema que explicamos en el siguiente apartado.

Borrados en cascada

Siguiendo con el ejemplo de 'Departamento-Empleado', si creamos un repositorio JPA con las operaciones básicas CRUD sobre 'Departamento', ¿Qué ocurrirá si hacemos un borrado de un departamento que tiene empleados asignados?

La respuesta es que, por defecto, se producirá una excepción, ya que, debido a la restricción de clave foránea, si se borrase solo el departamento, la base de datos quedaría inconsistente: habría empleados con un departamento inexistente asignado. Para solucionar esta situación tenemos varias opciones.

- 1) Borrado en cascada. Al borrar un departamento se borrarían automáticamente todos sus empleados. Se especifica junto al mapeo de la asociación:

```
@ManyToMany
@OnDelete (action = OnDeleteAction.CASCADE)
private Departamento departamento;
```

En apartados posteriores veremos la diferencia entre `@OnDelete` y `CascadeType.REMOVE`.

- 2) Fijar a `null` el valor de departamento en la tabla de empleados para aquellos empleados que pertenezcan al departamento eliminado. A nivel de integridad de base de datos, no es de las mejores opciones y además Hibernate no lo permite, para hacer esto deberíamos crear el esquema por fuera de Hibernate y añadir el atributo `ON DELETE SET NULL` a la restricción de clave foránea.
- 3) Una última opción sería verificar antes del borrado que no hay empleados asignados a ese departamento. Podríamos crear en el repositorio de *Empleado* un método para este fin:

```
@Query("select count (e) from Empleado e where e.departamento.id = ?1")
Long cantidadEmpleadosDepto(Long idDepto);
```

Y luego, en el servicio de Departamento, en el método de borrado de departamentos, antes de hacer el borrado, verificar que no hay empleados en ese departamento. Necesitaríamos injectar el repositorio de Empleados en este servicio, para poder invocar al método que acabamos de crear:

```
public void borrar(Long id) {
    Long cantEmpleadosDepto=empleadoRepository.cantidadEmpleadosDepto(id);
    if (cantEmpleadosDepto == 0) departamentoRepository.deleteById(id);
}
```

@OneToMany

Esta asociación es la inversa a la anterior, nos permite enlazar dos entidades, pero añadiendo la anotación a la entidad "uno". Para ello, en esa clase, además de `@OneToMany` incluiremos una colección de elementos "muchos".

Siguiendo con el ejemplo anterior de los empleados, podríamos pensar en una entidad que fuesen las *nóminas* que cobra cada empleado, que tiene claramente una relación "muchos a uno" con empleado: un empleado tiene muchas nóminas (una cada mes) pero una nómina solo pertenece a un empleado. Podríamos modelarla con una anotación `@ManyToOne` en la entidad Nómina, pero vamos a hacerla con `@OneToMany` para ver sus diferencias.

Primero creamos la entidad "*Nómina*" con sus atributos; como es la parte "muchos" no será necesario hacer nada más.

```
@Entity
public class Nomina {
    @Id
    @GeneratedValue (strategy=GenerationType.IDENTITY)
    private Long id;

    @DateTimeFormat(pattern = "yyyy-MM-dd")
    LocalDate fechaNomina;

    Double importeBruto;
    Double porcentImpuestos;
    Double importeNeto;
}
```

Ahora, en la entidad Empleado, añadiremos a sus los atributos previos, la anotación `@OneToMany` y la colección de nóminas.

```
@Entity
public class Empleado {
    @Id
    @GeneratedValue (strategy=GenerationType.IDENTITY)
    private Long id;
    ...
    @OneToMany(fetch = FetchType.EAGER, cascade = CascadeType.ALL)
    private List<Nomina> nominas = new ArrayList<>();
}
```

Expicaremos al final del capítulo el atributo *fetch* y el atributo *cascade* incluidos en la anotación.

Aunque es típico que la colección empleada sea un `ArrayList` puede ser también una colección de tipo conjunto (Set) e Hibernate lo recomienda por conseguir mayor eficiencia. Recordemos que los conjuntos no tienen posiciones como tal y no admite duplicados.

```
private Set<Nomina> nominas = new HashSet<>();
```

Relaciones bidireccionales

Si la asociación `@OneToMany` entre dos entidades no tiene la correspondiente `@ManyToOne` entre ellas misma en sentido contrario, decimos que es **unidireccional**, y es tal cual la acabamos de definir. En caso de que sí exista la complementaria `@ManyToOne` en la otra entidad, decimos que es **bidireccional**.

Siguiendo nuestro ejemplo, añadiríamos una asociación `@ManyToOne` en la clase Nómina similar a la que hicimos entre Empleado y Departamento en el apartado anterior, y en la `@OneToMany` que estábamos desarrollando en este apartado añadiríamos una nueva referencia a la que acabamos de añadir con el atributo ***mappedBy***.

La entidad Empleado quedaría así:

```

@Entity
public class Empleado {
    @Id
    @GeneratedValue (strategy=GenerationType.IDENTITY)
    private Long id;

    //resto de métodos, atributos y relaciones

    @ToString.Exclude
    @OneToMany(fetch = FetchType.EAGER, cascade = CascadeType.ALL,
               mappedBy="empleado") // orphanRemoval = true
    private List<Nomina> nominas = new ArrayList<>();
}

```

Y Nómina así:

```

@Entity
public class Nomina {
    @Id
    @GeneratedValue (strategy=GenerationType.IDENTITY)
    private Long id;
    . . .
    @ManyToOne
    @OnDelete (action = OnDeleteAction.CASCADE)
    private Empleado empleado;
}

```

Aspectos importantes de las direcciones bidireccionales

- Al tener la *List "nominas"*, mediante su *getter* podemos acceder a todas las nóminas de un empleado mediante un simple *getNominas()* sin necesidad de métodos adicionales en el repositorio de nóminas (si fuese solo unidireccional, necesitaríamos un método como: *findByEmpleado(Empleado empleado* en el repositorio de nóminas).
- Gracias a la anotación **CascadeType.ALL** Las relaciones se mantendrán sincronizadas cuando empleemos los métodos típicos de los repositorios como JpaRepository: save, remove, etc. Es decir, al guardar un empleado se guardarán todas sus nóminas, al borrar un empleado se borrarán sus nóminas. Puedes probar lo siguiente:

```

//creamos un empleado con id 1, nombre pepe y colección de nominas a null
Empleado empleado = new Empleado (1, "pepe", resto atributos , null);
//añadimos una nómina a la colección
empleado.getNominas().add(new Nomina(1, "2024-11-30",15000d, atributos ));
empleado.getNominas().add(new Nomina(2, "2024-12-31",2000d, atributos ));
//guardamos el empleado
empleadoRepository.save(empleado);

```

Si comprobamos el resultado en la base de datos, veremos una fila nueva en la tabla de Empleado pero también dos nuevas filas en la tabla de nóminas, esto sin haber invocado al método *save* del repositorio de nóminas.

- Incluimos la anotación **@ToString.Exclude** a la anotación en Empleado ya que un empleado referencia a sus nóminas, pero a su vez, cada nómina referencia a su empleado que, de nuevo, tiene muchas nóminas, etc, etc. etc....se puede producir un bucle infinito. Para arreglarlo basta con añadir esa anotación Lombok para cortar esa recursividad. **Habrá que añadirla en todas las relaciones bidireccionales que pudiesen generar esa recursividad.** Se puede poner en cualquiera de los dos extremos de la relación (o bien en la Empleado o bien en Nomina).

- Hibernate recomienda en su cuaderno de buenas prácticas establecer las relaciones 1 a N de forma bidireccional, ya que facilita la navegación por sus atributos en ambos sentidos. Sin embargo, habrá casos en que no sea lo más aconsejable, por ejemplo, si hubiese millones de nóminas para un mismo empleado, la colección tendría un tamaño muy grande y podría no ser muy eficiente. En estos casos es mejor plantear una sola relación de las que vimos en el apartado anterior: `@ManyToOne`.

Añadir la relación a la aplicación

Para incorporar a nuestra aplicación esta nueva entidad *Nómina* y su relación bidireccional con *Empleado* repetiríamos unos pasos similares a los empleados para *Departamento*, a saber:

- 1) Crear la entidad *Nómina* con la relación `@ManyToOne`.
- 2) Añadir a la entidad *Empleado* la relación `@OneToMany` con *mapped* si queremos que sea bidireccional, tal y como hemos mostrado previamente.
- 3) Crear un repositorio, servicio y controlador para *Nómina* similar al de departamento. Como comentamos más arriba, no es necesario en el repositorio ningún método para obtener las nóminas de un empleado concreto ya que lo haremos con el `getNominas()` del empleado.
- 4) El repositorio y el servicio de *Empleado* no se ven afectados.
- 5) En el controlador de *Nomina*, en el alta y edición de una nómina, hay que poder seleccionar el empleado: o bien mediante una lista de empleados inyectada en el controlador o bien recibiendo ya el empleado al que se le aplicará en el mapping (por ejemplo, a través de `@PathVariable`).

```
@PostMapping ("/nomina/nueva/submit/{idEmpleado}")
public String addNomina (@PathVariable Long idEmpleado, Nomina nomina){
```

- 6) Añadir las nuevas vistas para el CRUD de nómina y también modificar la vista de un empleado para poder acceder a sus nóminas.
- 7) Modificar el menú general para añadir una entrada nueva para la gestión de nóminas.
- 8) En el *CommandLineRunner*, el constructor de *Empleado* ahora deberá tener un parámetro más para la lista de nóminas (podemos ponerlo inicialmente a `null`).

CascadeType.ALL vs *CascadeType.REMOVE* vs *OnDeleteAction.CASCADE* vs *orphanRemoval=true*

Cuando tenemos dos entidades y de ellas depende de la otra, *CascadeType* hace referencia a todas las operaciones a realizar "en cascada" sobre la entidad dependiente cuando se produce un cambio en una entidad principal. Un caso típico es el borrado, en el ejemplo que estamos viendo *Empleado-Nomina*, al borrar un empleado, se borrarán todas sus nóminas.

Este comportamiento se gestiona mediante los valores asignados al atributo *CascadeType* de forma que, si toma el valor *REMOVE*, solo los borrados se propagarán en cascada, y si toma el valor *ALL* será cualquier operación la que se propagará.

OnDeleteAction.CASCADE realiza la misma operación que *CascadeType.REMOVE* (borrado en cascada para las entidades relacionadas) pero actúan sobre los extremos opuestos de la relación.

Se ve más fácil con un ejemplo: Si tenemos una relación muchos a uno entre *Empleado* y *Nomina* (un empleado puede tener varias nóminas, pero una nómina solo pertenece a un empleado), añadiendo la anotación de la relación el atributo `@OnDelete`, por ejemplo, así:

```

@Entity
public class Nomina {
    @Id
    private Long id;
    // resto de atributos y métodos
    @ManyToOne
    @OnDelete (action = OnDeleteAction.CASCADE)
    private Empleado empleado;

}

```

al borrar un empleado, se borrarán todas sus nóminas. Por el contrario, si incluyésemos en esa entidad CascadeType.REMOVE, al borrar una nómina, se borraría su empleado, algo que no es lo deseado.

```

@Entity
public class Nomina {
    @Id
    private Long id;

    @ManyToOne (cascade = CascadeType.REMOVE)
    private Empleado empleado;
    // resto de atributos y métodos
}

```

Lo habitual es usar CascadeType.REMOVE en las relaciones inversas, las `@OneToMany`, de forma que finalmente esta relación quedaría así:

```

@Entity
public class Empleado {
    @Id
    private Long id;

    // resto de atributos y métodos

    @OneToMany(cascade = CascadeType.ALL) // o bien CascadeType.REMOVE
    private List<Nomina> nominas = new ArrayList<>();
}

```

Así, al borrar empleado se borrarían todos sus nóminas y no al revés. Esto sí es lo habitual.

Para terminar, tenemos el atributo `orphanRemoval` que hace algo parecido a CascadeType.REMOVE, aunque más bien lo complementa. Supongamos que modificamos las nóminas que tiene asignados un empleado. Podría darse el caso de que quedasen nóminas "huérfanas", es decir, que no son referenciadas por nadie. Este atributo marcado a `true` los eliminaría (si no lo incluimos, su valor por defecto es `false`).

```

@Entity
public class Empleado {
    @Id
    private Long id;

    @OneToMany(cascade={CascadeType.ALL}, orphanRemoval=true)
    private List<Telefono> telefonos = new ArrayList<>();

    // resto de atributos y métodos
}

```

Opciones CascadeType

Como complemento al punto anterior, vamos a ver las opciones que tiene este parámetro y que, como ya hemos visto, hace referencia a cómo los cambios de estado que realizamos en una entidad se propagan a sus entidades dependientes. Sus posibles valores serían:

- **Persist y Merge:** las operaciones de guardado en las entidades padre se propagarán a las entidades relacionadas.
- **Remove:** elimina las entidades relacionadas cuando la entidad propietaria se elimina.
- **Refresh y Detach:** poco habituales, las dejamos fuera del curso.
- **All:** se aplican todos los tipos

@ManyToMany

Como su nombre indica, en este tipo de asociaciones una o varias instancias de una entidad pueden relacionarse con una o muchas de la otra entidad. Siguiendo con el ejemplo de empleados, podríamos tener una entidad *Proyecto* y decir que un empleado puede colaborar en varios proyectos y que en un proyecto colaboran varios empleados.

Estas asociaciones muchos a muchos necesitan una tabla que realice de enlace entre ambas entidades asociadas. También disponen de un tratamiento unidireccional y bidireccional.

@ManyToMany unidireccional

Debemos definir cuál de las entidades es la propietaria, y en ella incluiremos la lista de elementos de la clase opuesta (como en el caso de @OneToMany).

En el caso del Empleado:

```
@Entity
public class Empleado {
    @Id
    @GeneratedValue (strategy=GenerationType.IDENTITY)
    private Long id;

    @ToString.Exclude
    @ManyToMany(fetch = FetchType.EAGER, cascade = CascadeType.ALL)
    private List <Proyecto> proyectos = new ArrayList<>();
    //mantenemos las asociaciones previas, constructor...
}
```

Y por otra parte tendremos la otra entidad sin atributos adicionales.

```
@Entity
public class Proyecto {
    @Id
    @GeneratedValue (strategy=GenerationType.IDENTITY)
    private Long id;
    private String nombre;
}
```

@ManyToMany bidireccional

Añadiremos en el lado no propietario, en este caso Proyecto, el atributo **mappedBy** y una colección para almacenar los elementos de la entidad opuesta, en este caso Empleado.

```
@Entity
public class Proyecto {
    @Id
    private Long id;
    @NotEmpty
    private String nombre;

    @ToString.Exclude
    @ManyToMany(mappedBy ="proyectos")
    private List <Empleado> empleados = new ArrayList<>();
}
```

@ManyToMany con atributos extra

Puede haber atributos que sean propios de la asociación, en nuestro ejemplo, el “puesto” dentro de un proyecto (*product owner, team leader, team member, etc.*) sería un atributo de este tipo, ya que un

el mismo empleado puede tener distintos puestos en distintos proyectos. Algunos autores llaman a este tipo de asociación con atributos: **“clase de asociación”**.

En general, este será el modelo que emplearemos para relaciones muchos a muchos, ya que, aunque en un principio parezca que no tenemos atributos extra, pueden aparecer más adelante y de emplear otro modelo, habría que rehacer de nuevo las relaciones.

Como este nuevo atributo “puesto” no es ni del empleado, ni del proyecto, no lo podemos colocar en ninguna de las dos entidades, por lo que tenemos que debemos generar una nueva entidad, a la que podemos llamar *EmpleadoProyecto*, y que contendrá los nuevos atributos extra.

```
@Entity
public class EmpleadoProyecto {
    @NotEmpty
    private String puesto;
    . . .
```

Ahora tenemos que tomar una decisión sobre la clave de esta nueva entidad, por una parte, podemos hacer como en entidades anteriores y generar un nuevo atributo al que llamaríamos algo como ‘id’ y lo anotaríamos con `@Id`. La otra opción sería que la clave estuviese formada por dos atributos, el ‘id’ de empleado y el ‘id’ de proyecto. Vamos a optar por la primera solución, por su sencillez, pero en un apartado posterior (bajo el epígrafe `@IdClass`) veremos la segunda aproximación.

Entonces, la clase de asociación finalmente quedaría así:

```
@Data
@NoArgsConstructor
@AllArgsConstructor
@Entity
public class EmpleadoProyecto {
    @Id
    @GeneratedValue
    private Long id;

    @ManyToOne
    @JoinColumn(name = "empleado_id")
    @OnDelete(action=OnDeleteAction.CASCADE)
    private Empleado empleado;

    @ManyToOne
    @JoinColumn(name = "proyecto_id")
    @OnDelete(action=OnDeleteAction.CASCADE)
    private Proyecto proyecto;

    private String puesto;
}
```

Opcionalmente, solo si deseamos que la relación sea bidireccional, añadiremos cada una de las entidades extremo de la relación, las asociaciones `@OneToMany` con la nueva entidad creada. Así nos quedarían entonces las otras dos clases:

Empleado:

```
@Entity
public class Empleado {
    @Id
    @GeneratedValue
    private Long id;

    //resto de atributos y métodos
    @ToString.Exclude
    @OneToMany(mappedBy = "empleado", cascade = CascadeType.ALL)
    private List<EmpleadoProyecto> empleadoProyecto = new ArrayList<>();
```

```

    }

```

Proyecto:

```

@Data
@AllArgsConstructor
@NoArgsConstructor
@EqualsAndHashCode(of = "id")
@Entity
public class Proyecto {
    @Id
    @GeneratedValue
    private Long id;
    //resto de atributos y métodos
    @ToString.Exclude
    @OneToMany(mappedBy = "proyecto", cascade = CascadeType.ALL)
    private List<EmpleadoProyecto> empleadoProyecto = new ArrayList<>();
}

```

Añadir la relación a nuestra aplicación:

Para incorporar a la aplicación la nueva entidad *Proyecto* y su relación de tipo *“muchos a muchos con atributos extra”* con *Empleado* repetiríamos los mismos pasos que en las relaciones anteriores, aunque la gestión de esta nueva asociación requerirá un tratamiento ligeramente distinto a nivel controlador y vistas. Sería así:

- 1) Crear la entidad *Proyecto* y *EmpleadoProyecto* según acabamos de describir con sus relaciones.
- 2) Crear un repositorio, servicio y controlador para *Proyecto*. Podemos copiarlos de los correspondientes a *Departamento* ya que son análogos.
- 3) Crear un repositorio para *EmpleadoProyecto*. Como al seleccionar un empleado vamos a querer ver sus proyectos y al seleccionar un proyecto queremos ver sus empleados asignados, incluiremos en este repositorio un par de métodos derivados por nombre para obtenerlo: *findByProyecto* y *findByEmpleado* respectivamente, también uno para obtener la instancia correspondiente a un empleado con un proyecto: *findByEmpleadoAndProyecto*. Recordemos que con este tipo de métodos no es necesario escribir su código, JPA lo hace por nosotros.

```

public interface EmpleadoProyectoRepository extends
    JpaRepository<EmpleadoProyecto, Long> {
    // métodos derivados de nombre
    List<EmpleadoProyecto> findByEmpleado (Empleado empleado);
    //también: List<EmpleadoProyecto> findByEmpleadoId (Long idEmpleado);
    List<EmpleadoProyecto> findByProyecto (Proyecto proyecto);
    EmpleadoProyecto findByEmpleadoAndProyecto (Empleado emp, Proyecto proy);
}

```

- 4) Crear un servicio para *EmpleadoProyecto* (clase e interfaz). Para simplificar no incorporamos el método de modificar (en caso de querer modificar una asignación de empleado a proyecto la eliminaríamos y haríamos una nueva). Tampoco es necesario un *findAll()* ya que nunca querremos ver todos los empleados-proyecto globalmente, serán los proyectos de un empleado o los empleados de un proyecto.

```

@Service
public class EmpleadoProyectoServiceImplBD implements EmpleadoProyectoService {
    @Autowired
    private EmpleadoProyectoRepository empleadoProyectoRepository;

    public EmpleadoProyecto obtenerPorId(Long id) {
        return empleadoProyectoRepository.findById(id).orElse(null);
    }
    public EmpleadoProyecto añadir(EmpleadoProyecto empleadoProyecto) {

```

```

        return empleadoProyectoRepository.save(empleadoProyecto);
    }
    public void borrar(EmpleadoProyecto empleadoProyecto) {
        empleadoProyectoRepository.delete(empleadoProyecto);
    }
    public List<EmpleadoProyecto> obtenerPorEmpleado (Empleado empleado) {
        return empleadoProyectoRepository.findByEmpleado(empleado);
    }
    public List<EmpleadoProyecto> obtenerPorProyecto(Proyecto proyecto) {
        return empleadoProyectoRepository.findByProyecto(proyecto);
    }
    public EmpleadoProyecto obtenerPorEmpleadoProyecto(Empleado e, Proyecto p) {
        return empleadoProyectoRepository.findByEmpleadoAndProyecto(e,p);
    }
}

```

Recordemos que es en el servicio donde incorporamos la lógica de negocio. Si por ejemplo hubiese un número máximo de miembros para un proyecto aquí es donde, antes de incorporar un empleado a un proyecto, deberíamos verificar si se supera ese máximo y en caso afirmativo, no proceder con la alta de la entidad.

- 5) Como en las asociaciones anteriores, el repositorio y el servicio de *Empleado* no se ven afectados. Tampoco el controlador de *Empleado* ya que el proyecto no será un atributo único de empleado, como habíamos hecho hasta ahora ya que puede un empleado tener varios proyectos asignados. Crearemos un nuevo controlador para *EmpleadoProyecto*.
- 6) El nuevo controlador *EmpleadoProyecto* es un poco distinto a los demás, ya que no responde a consultas globales de todos los empleados o de todos los proyectos: consultaremos los proyectos de un determinado empleado o los empleados de un determinado proyecto. Obviamente también tendrá un método para crear nuevas instancias de *EmpleadoProyecto* y para eliminarlas.

```

@Controller
@RequestMapping("/emplProy")
public class EmpleadoProyectoController {

    @Autowired
    private EmpleadoProyectoService empleadoProyectoService;

    @Autowired
    private EmpleadoService empleadoService;

    @Autowired
    private ProyectoService proyectoService;

    @GetMapping("/emp/{id}") // Lista de proyectos de un empleado
    public String showProjectsByEmpl(@PathVariable long id, Model model) {
        Empleado e = empleadoService.obtenerPorId(id);
        model.addAttribute("listaEmpleadoProyecto",
                           empleadoProyectoService.obtenerPorEmpleado(e));
        model.addAttribute("empleado", empleadoService.obtenerPorId(id));
        return "empleadoProyecto/empListView";
    }
    @GetMapping("/pro/{id}") // Lista de empleados de un proyecto
    public String showEmplbyProject(@PathVariable long id, Model model) {
        Proyecto p = proyectoService.obtenerPorId(id);
        model.addAttribute("listaEmpleadoProyecto",
                           empleadoProyectoService.obtenerPorProyecto(p));
        model.addAttribute("proyecto", proyectoService.obtenerPorId(id));
        return "empleadoProyecto/proListView";
    }
}

```

Faltarían los métodos de borrado y alta:

```

@GetMapping("/delete/{id}")
public String showDeleteEmpl(@PathVariable long id) {
    empleadoProyectoService.borrar(empleadoProyectoService.obtenerPorId(id));
    return "redirect:/";
}

@GetMapping("/new")
public String showNewProjectEmpl(Model model) {
    model.addAttribute("empleadoProyectoForm",
        new EmpleadoProyecto());
    model.addAttribute("listaEmpleados",
        empleadoService.obtenerTodos());
    model.addAttribute("listaProyectos",
        proyectoService.obtenerTodos());
    return "empleadoProyecto/empProyNewFormView";
}

@PostMapping("/new/submit")
public String showNewProjectEmplSubmit(
    @Valid EmpleadoProyecto empleadoProyectoForm,
    BindingResult bindingResult) {
    if (!bindingResult.hasErrors())
        empleadoProyectoService.añadir(empleadoProyectoForm);
    return "redirect:/";
}

```

- 7) Modificar la vista inicial (la de listado de empleados) del proyecto para incluir un enlace a la vista del punto anterior que permite el CRUD de Proyectos.

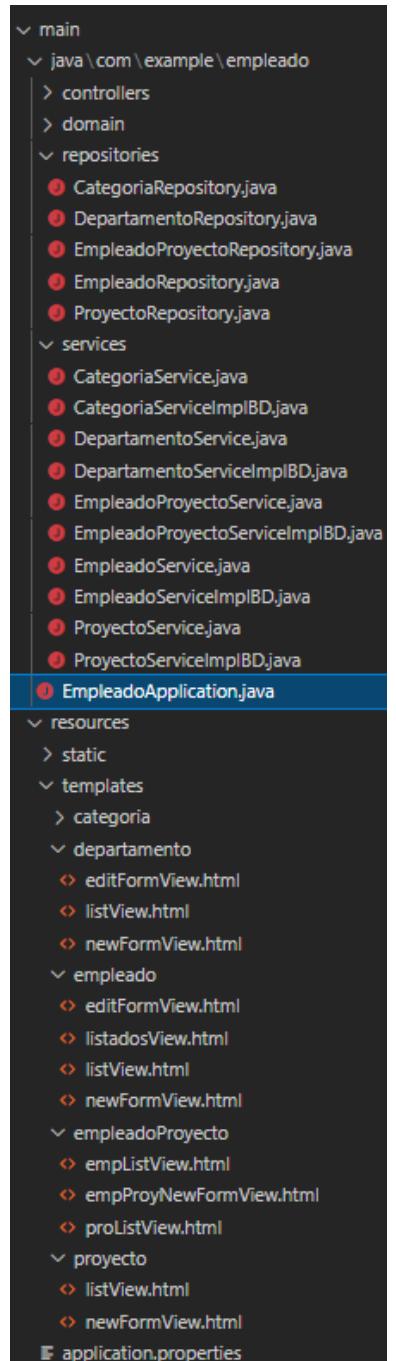
```
<a th:href="@{/proy/}">Gestión de proyectos</a><br/>
```

- 8) Modificamos esa vista de empleados para añadir un enlace en cada línea de empleado que nos lleve a una vista con los proyectos en los que está involucrado.

```


| ID | Nombre | Email | Salario | Activo | Genero | Imagen | Deptos | Proyectos |                                                                                                                                                                                            |
|----|--------|-------|---------|--------|--------|--------|--------|-----------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| Id | nombre | email | 0       | bool   | genero |        |        |           | <a th:href="@{/edit/{id}(id=\${empleado.id})}">Editar</a> <a th:href="@{/delete/{id}(id=\${empleado.id})}">Borrar</a> <a th:href="@{/emplProy/emp/{id}(id=\${empleado.id})}">Proyectos</a> |


```



- 9) Creamos una vista para lista de proyectos (listView.html), en principio similar a la de categorías o departamentos, pero debemos hacer algo análogo a lo que hemos realizado en la vista de empleados, añadiendo un enlace en cada línea de proyecto que nos lleve a una vista con los empleados que pertenecen a ese proyecto. Las vistas las organizamos en carpetas según la imagen lateral de la página anterior.

```

<h1>Listado de Proyectos</h1>
<table>
<thead><tr><th>ID</th><th>Nombre</th><th>Editar</th>
<th>Borrar</th><th>Empleados</th></tr></thead>

<tbody>
<tr th:each="proyecto : ${listaProyectos}">
<td th:text="${proyecto.id}">Id</td>
<td th:text="${proyecto.nombre}">nombre</td>
<td><a th:href="@{/proy/edit/{id}(id=${proyecto.id})}">
    Editar</a></td>
<td><a th:href="@{/proy/delete/{id}(id=${proyecto.id})}">
    Borrar</a></td>
<td><a th:href="@{/emplProy/pro/{id}(id=${proyecto.id})}">
    Empleados</a></td>
</tr>
</tbody>
</table>
<a th:href="@{/}">Inicio</a><br/>
<a th:href="@{/proy/new}">Nuevo Proyecto</a><br/>

```

- 10) Faltaría también la vista para el formulario de nuevo proyecto (y para editar proyecto):

```

<h1>Nuevo Proyecto</h1>
<form method="post" action="#">
    <th:action="@{/proy/new/submit}" th:object="${proyectoForm}">
        <label>Id:-solo lectura-<input type="text" readonly
            id="id" th:field="*{id}" /> </label><br/>
        <label>Nombre:<input type="text" id="nombre"
            th:field="*{nombre}" /></label><br/>
        <input type="submit" value="Enviar" />
    </form>
    <a th:href="@{/proy/}">Lista de proyectos</a><br/>

```

y nueva relación empleado-proyecto (*emplProyNewFormView.html*)

```

<h1>Empleado/Proyecto Nueva relación</h1>
<form method="post" action="#" th:action="@{/emplProy/new/submit}">
    <th:object="${empleadoProyectoForm}">
        Empleado:<br/>
        <select name="list1" th:field="*{empleado}">
            <option value="0">select option</option> <!--por defecto-->
            <option th:each="empleado:${listaEmpleados}"
                th:value="${empleado.id}" th:text="${empleado.nombre}"></option>
        </select><br/>

        Proyecto:<br/>
        <select name="list2" th:field="*{proyecto}">
            <option value="0">select option</option><!-- por defecto-->
            <option th:each="proyecto:${listaProyectos}"
                th:value="${proyecto.id}"
                th:text="${proyecto.nombre}"></option>
        </select><br/>
        <label>Puesto:<input type="text" id="puesto" th:field="*{puesto}" /></label><br/>
        <input type="submit" value="Enviar" />
    </form>

```

```
<a th:href="@{/}">Inicio</a><br/>
```

- 11) Finalmente necesitaríamos dos vistas para mostrar la relación empleado-proyecto. La primera sería la que muestra todos los proyectos de un empleado: *empleadoProyecto/empListView.html*, con opción de borrado y nueva relación:

```
<h1>Listado de Proyectos de un empleado</h1>
<h2 th:text='Empleado: ' + ${empleado.nombre}">Empleado</h2>
<table>
  <thead><tr>
    <th>ID</th><th>Nombre</th><th>Puesto</th><th>Borrar</th></tr>
  </thead>
  <tbody>
    <tr th:each="empleadoProyecto : ${listaEmpleadoProyecto}">
      <td th:text="${empleadoProyecto.proyecto.id}">Id</td>
      <td th:text="${empleadoProyecto.proyecto.nombre}">nombre</td>
      <td th:text="${empleadoProyecto.puesto}">Puesto</td>
      <td><a th:href="@{/emplProy/delete/{id}">Borrar</a></td>
    </tr>
  </tbody>
</table>
<a th:href="@{/}">Inicio</a><br/>
<a th:href="@{/emplProy/new}">Añadir empleado a proyecto</a><br/>
```

También incluimos la vista que muestra todos los empleados de un proyecto *empleadoProyecto/proListView.html*.

```
<h1>Listado de Empleados de un proyecto</h1>
<h2 th:text='Proyecto: ' + ${proyecto.nombre}">Proyecto</h2>
<table>
  <thead><tr>
    <th>ID</th><th>Nombre</th><th>Puesto</th><th>Borrar</th></tr>
  </thead>
  <tbody>
    <tr th:each="empleadoProyecto : ${listaEmpleadoProyecto}">
      <td th:text="${empleadoProyecto.empleado.id}">Id</td>
      <td th:text="${empleadoProyecto.empleado.nombre}">nombre</td>
      <td th:text="${empleadoProyecto.puesto}">Puesto</td>
      <td><a th:href="@{/emplProy/delete/{idEmpl}/{idProy}">Borrar</a></td>
    </tr>
  </tbody>
</table>
<a th:href="@{/}">Inicio</a><br/>
<a th:href="@{/emplProy/new/}"> Añadir empleado a proyecto </a><br/>
```

- 12) En el *CommandLineRunner* podemos incluir algún proyecto para tener datos iniciales.

```
proyectoService.añadir(new Proyecto(0L,"Nueva normativa UE"));
proyectoService.añadir(new Proyecto(0L,"Mejora Web actual"));
```

Si ejecutamos la aplicación, esta sería la estructura de tablas que nos mostraría la consola de H2:

jdbc:h2:mem:EmpleadoBD

SELECT * FROM EMPLEADO_PROYECTO;

ID	PUESTO	EMPLEADO_ID	PROYECTO_ID
1	Team Leader	1	1

(1 row, 4 ms)

@OneToOne

Son asociaciones similares a las `@OneToMany`, pero en el extremo en el que antes teníamos una colección (para la parte “many”), ahora tendremos una única instancia. Al igual que aquellas, estas relaciones también pueden ser unidireccionales o bidireccionales.

Siguiendo con nuestro ejemplo, pensemos en una entidad *Coche* (con id, matrícula, modelo, etc) que tendría una relación 1 a 1 con el empleado ya que un empleado puede tener asignado un coche o no tenerlo, pero nunca más de uno.

Uno de los lados es el propietario, en este caso el *Empleado*:

```
//anotaciones Lombok
@Entity
public class Empleado {
    @Id
    @GeneratedValue (strategy=GenerationType.IDENTITY)
    private Long id;

    private String nombre;

    //En caso de querer relación bidireccional añadiríamos:
    @OneToOne (mappedBy = "empleado")
    @ToString.Exclude
    private Coche coche;
}
```

y el otro no es propietario, en este caso la entidad *Coche*:

```
//anotaciones Lombok
@Entity
public class Coche {
    @Id
    @GeneratedValue (strategy=GenerationType.IDENTITY)
    private Long id;

    private String matricula;
    private String modelo;
    @OneToOne
    private Empleado empleado;
}
```

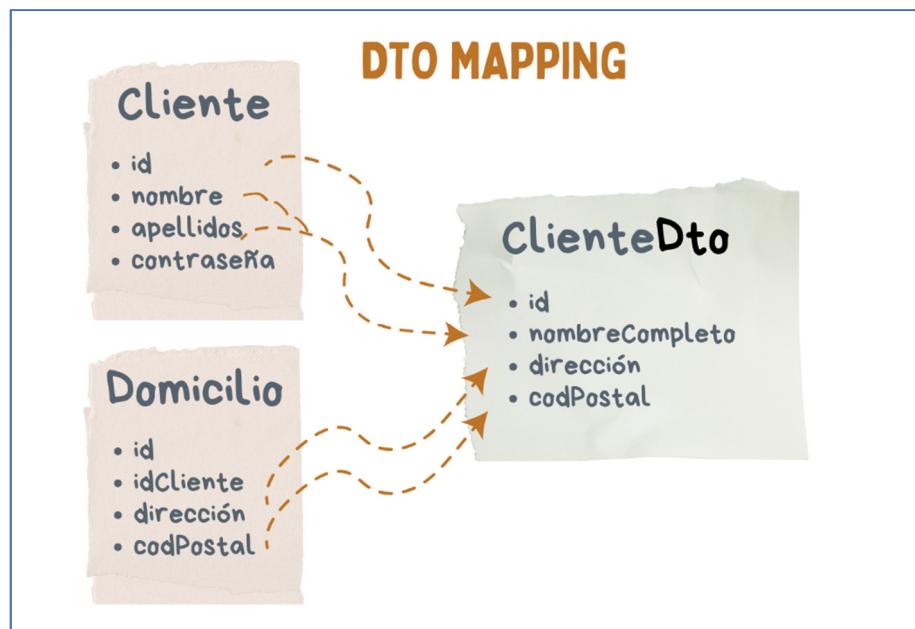
A tener en cuenta:

- Pensando en las vistas de usuario, será en el alta/edición de coches donde le introduciremos opcionalmente el empleado asignado a ese coche y no al revés, no habrá atribución de coche en el alta/edición de empleado.
- Pensando en la lógica de negocio, la matrícula de un coche debe ser única, por lo que en el alta/edición de un coche habrá que asegurarse de que la matrícula introducida no está asignada a otro coche
- También, al relacionar un coche con un empleado, debemos verificar que dicho empleado no esté asignado en ningún otro coche.
- Otro aspecto a resaltar es que en este caso no hemos añadido opciones de borrado en cascada, ya que al eliminar un coche no necesariamente hay que eliminar su empleado (a ese empleado se le puede asignar otro coche) y viceversa (al eliminar un empleado, el coche queda a disposición de otro empleado). Esto provocará un error si intentamos hacer un borrado de empleado con coche asignado, ya que el id de Empleado es clave foránea de Coche. Antes de borrar un empleado debemos asegurarnos de que no tiene coche asignado.

DTO

Una problemática recurrente cuando desarrollamos aplicaciones, es el diseño de la forma en que la información debe viajar desde la capa del controlador hasta las vistas (u hacia otros "clientes"), ya que muchas veces por desconocimiento o pereza, utilizamos las clases del modelo de dominio para enviar los datos, lo que puede ocasionar que retornamos más datos de los necesarios ([¿problemas de seguridad?](#)) o bien que tengamos que hacer varios envíos (uno por cada entidad) o bien datos con una estructura más compleja.

El patrón DTO tiene como finalidad de crear un objeto plano (POJO) con los atributos necesarios para enviar al cliente en una sola operación. Es frecuente que un DTO contenga información de múltiples entidades o tablas y concentrarlas en una única clase simple. Ejemplo:



Para construir un DTO podemos hacerlo de dos formas: o bien "a mano", a través de los setters del DTO empleando los getters de las clases origen de datos (no será mala opción para conjuntos sencillos), o bien crearlos de forma casi automática con la clase ModelMapper que veremos a continuación.

A nivel proyecto, los DTO pueden estar “cerca” de la carpeta (paquete) del modelo de dominio. O bien en la misma o como subcarpeta.

ModelMapper

ModelMapper, como acabamos de comentar, nos ayuda a transformar una o más entidades de nuestro modelo de negocio en un DTO, evitando tedioso código repetitivo. A nivel interno va a seguir unas reglas “inteligentes” de forma que a partir de los nombres que tengan los atributos del DTO y basándose en la entidad origen y sus asociaciones con otras entidades, haga el mapeo de atributos de forma automática y transparente para nosotros. Esto sin realizar configuración alguna, aunque también nos permitiría customizar su comportamiento de forma más precisa.

Para emplear ModelMapper debemos:

1. Añadir su dependencia al pom.xml del proyecto.

```
<dependency>
    <groupId>org.modelmapper</groupId>
    <artifactId>modelmapper</artifactId>
    <version>3.2.0</version>
</dependency>
```

2. Debemos crear un “bean” de tipo ModelMapper para toda nuestra aplicación, para poder injectar una instancia e invocar a sus métodos allá donde los necesitemos. Podemos hacerlo en una clase de configuración en archivo independiente: *ModelMapperConfig.java*:

```
@Configuration
public class ModelMapperConfig {
    @Bean
    ModelMapper modelMapper (){
        return new ModelMapper();
    }
}
```

3. La clase DTO será una clase sencilla con getters y setters y los atributos que queramos. El nombre del DTO debería contener el nombre de la clase de la que tomamos sus atributos y puede incluir atributos de otras clases asociadas, anteponiendo el nombre de la clase al del atributo.

Ejemplo:

```
@Getter @Setter
public class EmpleadoDTO {
    private Long id;
    private String nombre;
    private String departamentoNombre;
}
```

4. El método de esta clase para generar un DTO se llama *map()* y se le pasa como parámetro el objeto principal del que obtendremos los datos y el tipo de clase que devolverá:

```
@Autowired
private ModelMapper modelMapper;
...
EmpleadoDTO empleadoDto = modelMapper.map (empleado, EmpleadoDTO.class);
```

La generación del DTO se puede hacer en diversos puntos de la aplicación. Un lugar típico sería el controlador, o bien un método dentro del propio servicio, que recibiese el objeto (o conjunto de objetos) y lo transformase en DTO.

En el siguiente ejemplo, obtenemos todos los empleados de la base de datos, y en vez de pasarlos a la vista, los convertimos al DTO creado y enviamos la lista de DTO.

En el controlador:

```
@GetMapping({ "/", "/list" })
public String showList(Model model) {
    List<Empleado> listaEmpleados = empleadoService.obtenerTodos();
    List<EmpleadoDTO> listaDTO = empleadoService.convertEmpleadoToDto(listaEmpleados);

    model.addAttribute("listaEmpleados", listaDTO);
    return "empleado/listView";
}
```

Y en el servicio:

```
@Autowired
private ModelMapper modelMapper;
. . .
public List<EmpleadoDTO> convertEmpleadoToDto(List<Empleado> listaEmpleados) {
    List<EmpleadoDTO> listaEmpleadoDTO = new ArrayList<>();
    for (Empleado empleado : listaEmpleados)
        listaEmpleadoDTO.add(modelMapper.map(empleado, EmpleadoDTO.class));
    return listaEmpleadoDTO;
}
```

En el tema de API Rest trataremos más a fondo los DTO.

Otros conceptos JPA importantes

En este último apartado del tema vamos a ver conceptos complementarios que o bien no han sido mencionados previamente, o bien los pasamos por alto para no complicar en exceso cada apartado.

Fetch Type

Existen distintos tipos de captura o búsqueda (Fetch Types) en función de cómo se efectúa la búsqueda de los datos cuando hablamos de relaciones entre entidades:

- **Lazy:** (*lectura demorada*) El dato no se solicita hasta que se referencia. Es decir, si la entidad tiene una lista para almacenar las instancias de la otra entidad con la que se relaciona, Hibernate va a esperar a que se haga una consulta sobre la lista para obtener los datos de la base de datos.
- **Eager:** los datos se consultan por adelantado.

Por defecto, JPA usa por defecto Lazy para @OneToMany y @ManyToMany y Eager para los otros dos. Esto tiene sentido, pues son esos dos tipos los que a más objetos conectan en el otro lado de la relación. Al ser “lazy” el comportamiento por defecto, no debemos hacer nada para declarar estas relaciones como Lazy. Si queremos que sean Eager (evitaremos problemas, pero con peor rendimiento) podemos especificarlo en la anotación de la relación: `@OneToMany (fetch = FetchType.EAGER)`

Problema con fetch “Lazy”: una vez una entidad contenedora ha sido desconectada (detached) del gestor de persistencia (por ejemplo, al enviarla de vuelta al código cliente que la solicitó), esta se enviará tal como esté en ese momento sin importar en qué estado estén sus relaciones que hayan sido marcadas como Lazy. Si una relación ha sido inicializada antes de desconectar la entidad del gestor de persistencia, podremos acceder a sus valores de forma normal; en caso contrario, la relación no apuntará a ningún objeto, y por tanto obtendremos una excepción al intentar manejarla.

Paginación

Cuando una consulta sobre un repositorio devuelve un volumen de datos elevado, puede ser interesante ir recuperándolos en subconjuntos más pequeños, y que, a través de distintos eventos (como puede ser la petición del usuario) vayamos recuperando cada parte del conjunto total de datos, de uno en uno. Esto es lo que se conoce por *paginación*.

A veces esto ocurre de forma transparente para nosotros, como cuando deslizamos el dedo por la pantalla del móvil para avanzar las fotos de Instagram. El móvil no recibe todas las fotos posibles, las va recibiendo desde el servidor en conjuntos pequeños. Si en un momento el servidor responde más lento es cuando podemos ver *relojitos* de espera...hasta que el servidor responda ;)

Spring tiene definida dentro de la jerarquía de interfaces de repositorio la interfaz: `PagingAndSortingRepository<T, ID>` que define métodos que facilitan la ordenación y paginado. Como la interfaz que venimos usando hasta ahora (`JpaRepository`) hereda de esta interfaz, podemos usar esos métodos directamente. Los dos métodos fundamentales que hereda son:

`Iterable<T> findAll (Sort sort)`: devuelve las entidades ordenadas según el objeto Sort pasado como parámetro. Ejemplo, para devolver los empleados ordenados por email descendente:

```
empleadoRepository.findAll(Sort.by(Sort.Direction.DESC, "email"));
```

`Page<T> findAll (Pageable pageable)`: devuelve un objeto Page con las entidades devueltas, de acuerdo con las restricciones proporcionadas por el objeto Pageable pasado como parámetro. Este objeto `Pageable` nos permite definir tres parámetros fundamentales en la paginación:

- Número de página que estamos tratando en esta petición (empezando en cero). Si la consulta devuelve 20 páginas, este parámetro los moveremos entre 0 y 19 para acceder a sus registros.
- Tamaño de la página, es decir, la cantidad de registros de cada conjunto devuelto
- Criterio de ordenación.

Sobre nuestra entidad de ejemplo `Empleado`, podemos definir un servicio con un método que acceda al repositorio y devuelva los empleados paginados de 10 en 10 ordenados por nombre.

```
import org.springframework.data.domain.Page;
import org.springframework.data.domain.PageRequest;
import org.springframework.data.domain.Pageable;
import org.springframework.data.domain.Sort;

@Service
public class EmpleadoService {

    @Autowired
    private EmpleadoRepository empleadoRepository;

    private final Integer pageSize = 10;

    public List<Empleado> getEmpleadosPaginados(Integer pageNum) {
        Pageable paging = PageRequest.of(pageNum, pageSize, Sort.by("nombre"));
        Page<Empleado> pagedResult = empleadoRepository.findAll(paging);
        if (pagedResult.hasContent()) return pagedResult.getContent();
        else return null;
    }
}
```

Consideraciones:

- No es necesario extender `PagingAndSortingRepository` ya que el que usamos habitualmente `JpaRepository` es hija de esta y por tanto incluye todos estos métodos de paginación.
- Desde el controlador debemos invocar a este método del servicio con diferentes valores de número de página, entre cero y el total de páginas de la consulta menos uno.
- El orden es un parámetro opcional, podríamos no informarlo y los mostraría desordenados.
- Si le añadimos el parámetro de ordenación, por defecto es ascendente. Hay que añadirle `.descending()` para que sea descendente.

- La clase `Page` nos ofrece dos métodos interesantes para saber el total de páginas y el total de elementos, respectivamente: `int getTotalPages()`, `long getTotalElements()`.

Puede ser interesante añadir un método al servicio para saber el total de páginas de la consulta, para que el controlador sea capaz de gestionar peticiones de páginas correctas e incorrectas:

```
public int getTotalPaginas() {
    Pageable paging = PageRequest.of(0, pageSize, Sort.by("nombre"));

    Page<Empleado> pagedResult = empleadoRepository.findAll(paging);
    return pagedResult.getTotalPages();
}
```

El controlador debe hacer la llamada al servicio para obtener la página de resultados que necesite y también puede obtener el total de páginas para controlar errores (esto es, solicitar una página mayor que el total de páginas) y también para pasarlo a la vista el valor de página siguiente y página anterior, para los enlaces que permitan al usuario navegar por el conjunto total de resultados:

```
@Controller
public class EmpleadoController {
    @Autowired
    private EmpleadoService empleadoService;

    @GetMapping("/")
    public String showList(@RequestParam(required = false) Integer pag, Model model) {
        int ultPag = empleadoService.getTotalPaginas() - 1;
        if (pag == null || pag < 0 || pag > ultPag) pag = 0;
        Integer pagSig = ultPag > pag ? pag + 1 : ultPag;
        Integer pagAnt = pag > 0 ? pag - 1 : 0;
        model.addAttribute("listaEmpleados", empleadoService.getEmpleadosPaginados(pag));
        model.addAttribute("pagSiguiiente", pagSig);
        model.addAttribute("pagAnterior", pagAnt);
        model.addAttribute("pagFinal", ultPag);
        return "listView";
    }
}
```

Y por último tendríamos la vista con el siguiente `<body>`:

```
<body>
    <h1>Listado de empleados</h1>
    <table>
        <thead><th>ID</th><th>Nombre</th><th>Email</th><th>Salario</th></thead>
        <tbody><tr th:each="empleado : ${listaEmpleados}">
            <td th:text="${empleado.id}">Id</td>
            <td th:text="${empleado.nombre}">nombre</td>
            <td th:text="${empleado.email}">email@gmail.com</td>
            <td th:text="${empleado.salario}">0</td>
        </tr>
    </tbody>
</table>
<a th:href="@{/?pag=0}">Pág inicial</a> | <a th:href="@{/(pag=${pagAnterior})}">Pág ant</a> | <a th:href="@{/(pag=${pagSiguiiente})}">Pág sig</a> | <a th:href="@{/(pag=${pagFinal})}">Ult.pág</a>
</body>
```



http://localhost:9000/?pag=6

Listado de empleados

ID	Nombre	Email	Salario
54	Mari Colquyte	mcolquyte1h@vistaprint.com	1883.0
6	Marina Simson	msimson5@wikipedia.org	2613.0
13	Marsh Dulanty	mdulantyc@vkontakte.ru	2934.0
23	Marwin Pabst	mpabstm@slashdot.org	2171.0
15	Mathew Keer	mkeere@microsoft.com	931.0
50	Maurise Eingerfield	meingerfield1d@house.gov	1799.0
97	Merilee Jeayes	mjeayes2o@tripod.com	1859.0
83	Merrile Kiddy	mkiddy2a@hud.gov	932.0
72	Micheline Chaldecott	mchaldecott1z@archive.org	2298.0
78	Milly Casol	mcasol25@icq.com	1287.0

[Pág.inicial](#) | [Pág.ant](#) | [Pág.sig](#) | [Utl.pág](#)

Herencia

Tanto en Hibernate como en JPA se definen tres estrategias para mapear esta relación de clases a tablas de nuestra base de datos:

- **Single Table:** una sola tabla para guardar toda la jerarquía de clases. Tiene la ventaja de ser la opción que mejor rendimiento da, ya que sólo es necesario acceder a una tabla (está totalmente desnormalizada). Tiene como inconveniente que todos los campos de las clases hijas tienen que admitir nulos, ya que cuando guardemos un tipo, los campos correspondientes a los otros tipos de la jerarquía no tendrán valor. Es la estrategia por defecto.
- **Joined:** Una tabla para el padre de la jerarquía, con los atributos comunes, y otra tabla para cada clase hija con los atributos concretos. Es la opción más normalizada, y por lo tanto la más flexible, ya que para añadir nuevos tipos basta con añadir nuevas tablas y si queremos añadir nuevos atributos sólo hay que modificar la tabla correspondiente al tipo donde se está añadiendo el atributo. Tiene la desventaja de qué, para recuperar la información de una clase, hay que ir haciendo join con las tablas de las clases padre.
- **Table per class:** Una tabla independiente para cada tipo. En este caso cada tabla es independiente, y los atributos del padre (atributos comunes en los hijos), tienen que estar repetidos en cada tabla. En principio puede tener problemas de rendimiento, por lo que es la menos recomendada.

Veamos un mismo ejemplo con las dos primeras estrategias. Supongamos una clase Vehículo (atributos: id y modelo) y dos clases hijas: Moto (atributo: potencia) y Coche (atributo: peso). En todos los casos tendremos tres archivos: *Vehiculo.java*, *Moto.java* y *Coche.java*.

Estrategia: Single Table

La superclase vendrá anotada con el tipo de estrategia empleada y un discriminador, que será finalmente una columna que nos indicará el tipo de hijo ('Moto' o 'Coche' en este caso):

```
@Entity
@Inheritance( strategy = InheritanceType.SINGLE_TABLE )
@DiscriminatorColumn( name="tipoVehiculo" )           //no incluimos este atributo
public class Vehiculo {
    @Id
    private int id;
    private String modelo;
    //resto de atributos, constructores y métodos
}
```

Y las clases hijas con el atributo @DiscriminatorValue que distingue cada tipo de hijo.

```
@Entity
@DiscriminatorValue( value="Moto" )
public class Moto extends Vehiculo {
    private int potencia;
    //resto de atributos, constructores y métodos
}
@Entity
@DiscriminatorValue( value = "Coche" )
public class Coche extends Vehiculo {
    private int peso;
    //resto de atributos, constructores y métodos
}
```

Quedando una tabla así:

id	Modelo	tipoVehiculo	potencia	peso
1	Yamaha XMax	Moto	30	
2	Seat Ibiza	Coche		1500
3	Kimko Like	Moto	10	
4	Opel Mokka	Coche		1100

Estrategia: Joined

La estructura de las clases sería exactamente igual salvo el modificador de la anotación de herencia:
`@Inheritance(strategy=InheritanceType.JOINED)`

Y que ya no es necesario `@DiscriminatorColumn` y `@DiscriminatorValue`

Obtendríamos las siguientes tablas:

Id	Modelo
1	Yamaha XMax
2	Seat Ibiza
3	Kimko Like
4	Opel Mokka

id	potencia
1	30
3	10

id	peso
2	1500
4	1100

Otras anotaciones

@Transactional

Una transacción es una operación atómica de base de datos, esto es, que se realiza completamente o no se realiza. Imagina un escenario de cuentas corrientes y qué tenemos un servicio con un método de traspaso de dinero entre cuentas: realizamos primero una operación que retira un determinado importe de una cuenta y justo a continuación la operación que ingresa ese importe en otra cuenta. ¿Qué ocurre si el método falla en mitad del proceso y solo se ha hecho la retirada, pero no el ingreso? La respuesta es que la base de datos quedaría inconsistente.

Anotando el método con `@Transactional`, o bien hará todas las operaciones o ninguna, es decir, si hubiese algún problema, hará *roll back* de las operaciones ya realizadas para volver a la situación inicial.

Podemos añadir la anotación a nivel servicio, y así afectaría a todos sus métodos. En operaciones de solo lectura, podemos añadirle a la anotación el atributo `readOnly = true` para mejorar el rendimiento.

`@Transactional(readOnly = true)`

@IdClass

En las relaciones muchos a muchos, que generaban una nueva entidad, comentábamos que teníamos dos opciones sobre la clave de esa nueva entidad, atributo 'id' o bien una clase formada por dos atributos, las claves de las entidades de los extremos, en nuestro caso el 'id' de empleado y el 'id' de proyecto. En los apartados previos optamos por la primera opción, vamos a ver ahora la segunda.

Básicamente, lo que necesitamos es crear una clase extra para la clave primaria de esta nueva entidad. Esto es necesario ya que la anotación `@Id` solo funciona como clave primaria para atributos simples. Esta clase extra que actuará como clave de la entidad asociación deberá cumplir las siguientes características:

- Debe ser una clase pública.
- Debe tener un constructor sin argumentos.
- Debe implementar Serializable
- No debe tener clave primaria propia.
- Debe implementar los métodos `equals` y `hashCode`.
- Contendrá como atributos las claves de las entidades implicadas (generalmente Long) pero le pondremos como nombre el de su entidad.

En nuestro caso esta clase extra será *EmpleadoProyectoId*.

```
//anotaciones Lombok
public class EmpleadoProyectoId implements Serializable {
    private Long empleado;
    private Long proyecto;
}
```

Ahora la clase de asociación *EmpleadoProyecto*, con el *@IdClass* creado en el paso anterior quedaría:

```
@Data
@NoArgsConstructor
@AllArgsConstructor
@Entity
@IdClass(EmpleadoProyectoId.class)
public class EmpleadoProyecto {

    @Id
    @ManyToOne
    @JoinColumn(name = "empleado_id")
    @OnDelete(action=OnDeleteAction.CASCADE)
    private Empleado empleado;

    @Id
    @ManyToOne
    @JoinColumn(name = "proyecto_id")
    @OnDelete(action=OnDeleteAction.CASCADE)
    private Proyecto proyecto;

    private String puesto;
}
```

@NaturalId

Es una anotación propia de Hibernate. Representa un identificador único que, si bien no es el mejor candidato para clave primaria, es conveniente indicárselo a Hibernate, ya que lo puede usar para hacer más eficiente. Podría ser el DNI en una entidad *Persona*, la matrícula en una entidad *Coche*. Podríamos usar este *NaturalId* en consultas y en métodos como *equals()* y *hashCode()*.

JPA nos ofrece métodos ya implementados como *findBySimpleNaturalId* al que le pasamos un valor del atributo anotado con *@NaturalId* y nos devolvería un *Optional* de la entidad con ese valor.

@Transient

Se asigna a atributos de una entidad y se utiliza para indicar que ese atributo no debe de ser persistente, de esta manera, JPA pasa por alto el atributo y no es tomado en cuenta a la hora de persistir la entidad en la base de datos.

En la práctica no es común utilizar esta anotación, debido a que las entidades, por lo general, solo tienen los atributos que mapean con la base de datos. Sin embargo, existen ocasiones en donde puede ser útil. Un ejemplo muy habitual es cuando agregamos un logger a la clase entidad. Esta instancia de *java.util.logging.Logger* será una propiedad más de la entidad, pero no deseamos que sea persistente, para lo cual lo marcamos con *@Transient*.

@Enumerated

Esta anotación se usa para definir como persistir enumeraciones de nuestro proyecto en la base de datos. Tenemos dos formas:

- **String:** persiste la enumeración por su nombre, lo que significa que será una columna alfanumérica. La anotación quedaría así: *@Enumerated(value = EnumType.STRING)*
- **Ordinal:** persiste un valor entero que corresponde al valor ordinal o posición de valor en la enumeración. La anotación quedaría así: *@Enumerated(value = EnumType.ORDINAL)*
-

Ejemplo:

```
@Entity
public class Empleado {
    @Id
    private Long id;
    private String nombre;
    @Enumerated(value = EnumType.STRING)
    private Genero genero;
    //resto de atributos, constructores y métodos
}
public enum Genero { MASCULINO, FEMENINO, OTROS };
```

@Temporal

Se usa para persistir las fechas en la base de datos de forma simple. Una de las principales complicaciones cuando trabajamos con fecha y hora es determinar el formato empleado por el manejador de base de datos. Mediante el uso de `@Temporal` es posible determinar si nuestro atributo almacena Hora, Fecha u Hora y fecha, y es posible utilizar la clase Date o Calendar para estos fines. Se pueden establecer tres posibles valores para la anotación:

- **DATE**: Acotara el campo solo a la Fecha, descartando la hora. `@Temporal(TemporalType.DATE)`
- **TIME**: Acotara el campo solo a la Hora, descartando a la fecha. `@Temporal(TemporalType.TIME)`
- **TIMESTAMP**: Toma la fecha y hora. `@Temporal(TemporalType.TIMESTAMP)`

Ejemplo:

```
@Entity
public class Empleado {
    @Id
    private Long id;
    private String nombre;
    @Temporal(TemporalType.DATE)
    private Date fechaNacimiento;
    //resto de atributos, constructores y métodos
}
```

@MapsId

En relaciones `@OneToOne`, las dos entidades involucradas tienen su propio atributo clave, anotado con `@Id`, como es una relación uno a uno, hay una vinculación directa entre uno y otro, por lo que no es lo más eficiente tener ambos identificadores. La anotación `@MapsId` sirve para indicarle a la entidad dependiente que use en el atributo `@Id` el mismo valor que en la entidad principal. En el ejemplo que habíamos visto en las relaciones `@OneToOne` entre `Empleado` y `Coche` sería:

```
@Entity
public class Coche {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;

    private String matricula;
    private String modelo;
    @OneToOne
    @JoinColumn(name = "empleado_id")
    @MapsId
    private Empleado empleado;
    //anotaciones Lombok + constructor sin id
}
```

Entity Manager

Esta es una interfaz fundamental de JPA, ya que se encarga de realizar todas las operaciones de persistencia sobre la base de datos. Se encarga de establecer la conexión transaccional con la base de datos, mantener en memoria una caché con las entidades que gestiona y es responsable de sincronizarlas correctamente con la base de datos cuando se realiza un *flush*. El conjunto de entidades que gestiona un entity manager se denomina su contexto de persistencia.

En nuestro caso, todo este proceso se está haciendo de forma transparente para nosotros, todo gestionado por los repositorios (por ejemplo, JpaRepository) pero podríamos crearlo nosotros mismos y usar sus métodos, por ejemplo *persist()* para guardar los datos, *remove()* para eliminar, etc. Ejemplo:

```
@Repository
public class miRepository {
    @Autowired
    private EntityManager entityManager;

    public void save (MiEntidad miEntidad) {
        entityManager.persist (miEntidad);
    }
}
```

Recuerda que en los recursos disponibles en el área de descargas hay un ejemplo de subida de ficheros al servidor (es un CRUD de una entidad llamada Cliente sobre H2) Ahora es buen momento para echarle un ojo y retomar el tema de la gestión de ficheros en el servidor.

Tema 8: Seguridad y Control de Acceso

Introducción

Entendemos por control de acceso tanto la **autentificación** (¿Quién eres?) como a la **autorización** (¿Qué permisos tienes?). En cuanto a la autentificación tendremos que encargarnos de distintas tareas como es el registro de usuarios, el proceso de login y logout de los mismos, etc. Una vez que un usuario esté identificado en la aplicación tendremos que gestionar sus permisos.

Otro concepto a tener en cuenta es de '**rol**'. Los roles son agrupaciones de permisos comunes a varios usuarios. Roles típicos son los usuarios registrados sin más, usuarios con más privilegios o usuarios con todos los privilegios (también llamados administradores), así hablaremos de roles como USER, MANAGER, ADMIN, etc. y cada usuario estará asignado a uno de estos roles.

Una forma sencilla de trabajar consistirá en establecer para cada ruta de mapping de controlador (que es la forma de acceder a los recursos de nuestras aplicaciones) qué roles, y por lo tanto qué usuarios, pueden acceder a esa ruta. Podremos incluso discriminar con qué verbo HTTP puede acceder y con cuál no.

Así pues, siguiendo con la aplicación ejemplo con Spring MVC que llevamos planteando en este manual, correspondiente a un CRUD de una entidad *Empleado*, podríamos tener un esquema que el que los usuarios con el rol USER pudiesen acceder al alta de empleados: `@GetMapping("new")` y `@PostMapping("/new/submit")`, pero no al borrado de los ya creados `@GetMapping("/delete/{id}")` que lo podrían hacer usuarios con otro rol, por ejemplo, los administradores. Los usuarios no identificados podrían consultar la lista de empleados, pero no hacer ningún cambio. Este sería un esquema posible, pero podríamos tener cualquier otro.

Sesiones

Como comentamos a principio de curso, el protocolo **HTTP es un protocolo sin estados**, por lo que cuando se envía una petición, no se "recuerda" nada de las anteriores, cada una es totalmente independiente. Esto representa un problema para la autentificación de usuarios, ya que al navegar por una aplicación que requiera estar identificados, deberíamos estar enviando nuestro usuario y contraseña en cada petición. Para solucionar esta situación están las sesiones.

Una **sesión** es un mecanismo que permite conservar información sobre un usuario al pasar de una petición a otra (de una página web a otra) dentro de un mismo sitio web. Los datos referentes a esa sesión se almacenan en el servidor. El cliente almacenará sus datos de sesión, generalmente en una cookie, y esa información se enviará en cada petición. Al llegar al servidor la petición, se compara el identificador de sesión que le llega en la cookie con la sesión guardada en el propio servidor, y así se autentifica al usuario en cada petición.

Cuando se cierre la sesión (por tiempo, ya que las sesiones tienen un tiempo de expiración, o porque el usuario se ha desconectado) se elimina la información de esa sesión en el servidor por lo que esa cookie ya no tendrá validez.

Este modelo de sesión+cookie es válido para las aplicaciones web clásicas, las que desarrollamos con Spring MVC, pero cuando hablamos de API Rest este modelo no es válido, ya que los clientes de nuestra aplicación serán variados y no necesariamente trabajarán con cookies. Para API Rest usaremos otros métodos como **JWT** (JSON Web Token), pero esto lo veremos en un apartado posterior. Por ahora nos centraremos en el modelo MVC.

En la siguiente figura podemos ver el esquema de funcionamiento del proceso de login y petición de usuario registrado en el sistema.



Configuración básica

Spring Framework posee un módulo llamado Spring Security que se encarga de todos los aspectos de seguridad y control de acceso de nuestras aplicaciones. Es un módulo muy amplio, tanto en securización ante posibles ataques como en el tratamiento del control de acceso.

Para incluir Spring Security en nuestros proyectos SpringBoot basta con incorporar la dependencia *starter-security* en el archivo pom.xml.

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

Por sólo incluir la dependencia, sin ninguna configuración adicional, de manera predeterminada obtenemos los siguientes beneficios:

- Previene distintos tipos de ataques CSRF (Session Fixation, X-XSS-Protection, etc.)
- Protege el acceso a la aplicación, impidiendo que ningún usuario no identificado pueda invocar a cualquier ruta de los métodos de nuestros controladores y por tanto acceder a la aplicación.
- Genera un formulario de login en la ruta: `/login`.
- Genera una página de logout en la ruta: `/logout`.
- Permite hacer login mediante el formulario anterior y permite logout en la ruta `/logout`.
- Registra un filtro llamado `springSecurityFilterChain` que se encarga de proteger las contraseñas, redirigir al formulario de login cuando es necesario, etc.



Si quieras profundizar en el tema de los ataques CSRF puedes ver este video: <https://www.youtube.com/watch?v=CXSE89JGnek&t=01m13s> que explica de forma muy clara conceptos como CSRF, cookies/sesiones y CORS. Merece la pena verlo!

Bastaría entonces con añadir en el archivo *application.properties* las dos líneas siguientes para que solo el usuario indicado pudiese acceder a la aplicación.

```
spring.security.user.name =user
spring.security.user.password =1234
```

Cuando el usuario intente acceder a cualquier URL de la aplicación, Spring Boot y el *Security Filter* de HTTP redirigirá al usuario al formulario de identificación */login*, donde solicitará al usuario a insertar el nombre y contraseña para proceder. Este formulario se crea de forma automática.

Este mecanismo tan restrictivo podría ser suficiente para pequeñas aplicaciones monousuario donde sólo se necesita restringir el acceso de forma general, pero queda muy reducido en aplicaciones donde hay secciones accesibles al público en general y otras protegidas, dependiendo de si el usuario está identificado o incluso solo si éste tiene un rol determinado.

Para comenzar a configurar la seguridad, eliminaremos las dos líneas anteriores de *application.properties* y crearemos una clase anotada con **@EnableWebSecurity** y **@EnableMethodSecurity** con la configuración deseada. Podemos crearla en cualquier punto debajo del paquete raíz de la aplicación, pero lo habitual será tenerla en un paquete */security* o en el paquete */config*.

Nota: en versiones anteriores de Spring se configuraban estos aspectos extendiendo la clase *WebSecurityConfigurerAdapter*, pero desde Spring 5.7 está marcada como *deprecated* y debemos evitar su uso.

```
@Configuration
@EnableWebSecurity
@EnableMethodSecurity
public class SecurityConfig {
    @Bean
    public AuthenticationManager authenticationManager(
        AuthenticationConfiguration authenticationConfiguration)
        throws Exception {
        return authenticationConfiguration.getAuthenticationManager();
    }
    @Bean
    public PasswordEncoder passwordEncoder() {
        return new BCryptPasswordEncoder();
    }
    @Bean
    public UserDetailsService users(PasswordEncoder passwordEncoder) {
        UserDetails user = User.builder()
            .username("user1")
            .password(passwordEncoder.encode("1234"))
            .roles("USER")
            .build();
        return new InMemoryUserDetailsManager(user);
    }
}
```

Este sería un primer ejemplo, con una sencilla autentificación de usuarios en memoria, con un solo usuario 'user1', que tendría un rol básico llamado 'USER' y contraseña encriptada '1234'. Más adelante sustituiremos los usuarios en memoria por usuarios reales en base de datos.

Debemos fijarnos en que creamos tres beans para controlar la seguridad:

- **AuthenticationManager**
- **PasswordEncoder**
- **UserDetailsService**

Podemos sustituir el `UserDetailsService` por otro que añada dos usuarios, el `user1` actual y añadir un nuevo usuario y rol (`admin1` / `ADMIN`):

```
@Bean
public UserDetailsService users(PasswordEncoder passwordEncoder) {
    UserDetails user = User.builder()
        .username("user1")
        .password(passwordEncoder.encode("1234"))
        .roles("USER")
        .build();
    UserDetails admin = User.builder()
        .username("admin1")
        .password(passwordEncoder.encode("1234"))
        .roles("USER", "ADMIN")
        .build();
    return new InMemoryUserDetailsManager(user, admin);
}
```

Ahora vendría la parte más interesante, en la que indicamos el comportamiento a nivel de acceso de las distintas rutas de los mappings de nuestros controladores, y por tanto definir quién puede acceder a qué parte de la aplicación. Lo haremos en el mismo archivo, en un nuevo `bean` llamado `SecurityFilterChain`.

```
@Bean
public SecurityFilterChain filterChain(HttpSecurity http) throws Exception {
    http.headers(headersConfigurer -> headersConfigurer
        .frameOptions(HeadersConfigurer.FrameOptionsConfig::sameOrigin));
    http.authorizeHttpRequests(
        auth -> auth
            .requestMatchers("/").permitAll()
            .requestMatchers("/new/**").hasAnyRole("USER", "ADMIN")
            .requestMatchers("/edit/**", "/delete/**").hasRole("ADMIN")
            .requestMatchers(PathRequest.toStaticResources().atCommonLocations()).permitAll()
            .requestMatchers("/h2-console/**").hasRole("ADMIN")
            .anyRequest().authenticated()
            .formLogin(formLogin -> formLogin
                .defaultSuccessUrl("/", true)
                .permitAll())
            .logout(logout -> logout
                .permitAll())
        // .csrf(csrf -> csrf.disable())
        .httpBasic(Customizer.withDefaults()); //import org.springframework.security.config
    http.exceptionHandling(exceptions -> exceptions.accessDeniedPage("/accessError"));
    return http.build();
}
```

Se puede configurar de distintas formas, pero esta sería la más sencilla. Para ello usamos el método `requestMatchers`, al que le pasamos como parámetro la ruta o rutas sobre las que gestiona su acceso y a continuación quién tiene permiso: `permitAll()`, `hasRole (ROLE)`, `hasAnyRole (ROLE1, ROLE2, ...)`, `denyAll()`, `authenticated()`, etc. Así pues:

- `requestMatchers (" / ").permitAll()` : acceso a todo el público a la ruta raíz.
- `requestMatchers (" /new/**").hasRole("USER")` : acceso a la ruta `/new` a aquellos usuarios autenticados y con rol "USER". Los dos asteriscos `/**` indican que se extienden esos permisos a las "subrutas" debajo de la ruta indicada, por ejemplo: `/new/submit`.
- `requestMatchers("/edit/**", "/delete/**").hasRole("ADMIN")` : acceso a las rutas `/edit` y `/delete` y las subrutas debajo de ellas solo para los administradores.
- `requestMatchers(PathRequest.toStaticResources().atCommonLocations()).permitAll()`: acceso a los recursos estáticos de la aplicación, con nombre estándar, esto es: `/css`, `/js`, `/images`.

- Una vez que terminamos con los *requestMatchers* incluimos una sola vez `anyRequest()` para indicar quién tiene acceso al resto de rutas, las no indicadas en los *requestMatchers*, en este caso, solo a usuarios autentificados (podría ser `permitAll()` para que fuesen públicas).
- A continuación, permitimos el acceso a la ruta de `/login` y `/logout` a todo el público.
- En la última línea (*exceptionHandling*) indicamos la ruta a la que redirigimos a los usuarios que no tengan permisos suficientes para acceder a algún determinado punto de la aplicación. Debemos añadir entonces un método en un controlador con `@GetMapping("/accessError")`; con su tratamiento.
- En este ejemplo solo hemos incluido las rutas, pero *requestMatchers* admite un formato en el que también se le pase el verbo HTTP sobre el que queremos trabajar:
`.requestMatchers(HttpMethod.POST, "/path").denyAll()`

Finalmente, en las vistas de la aplicación deberemos incluir los enlaces para login y logout. Los controladores y vistas para estos enlaces los genera Spring Security por nosotros:

```
<a th:href="@{/login}">Login</a> | <a th:href="@{/logout}">Logout</a>
```

A tener en cuenta:

- **Las reglas de los *requestMatchers* se evalúan por orden:** en cuanto una ruta encaja, se determinan sus permisos en función de esa línea y ya no se evalúan las siguientes, por lo que es importante el orden en el que se sitúan. Por ejemplo, si la primera fuese `.requestMatchers("/**").permitAll()` ya no tendría sentido añadir ninguna más a continuación, ya que habría un acceso libre a la aplicación completa.
- Hay que tener cuidado de no repetir la misma ruta en dos *requestMatchers*, ya que solo se tendría en cuenta el primero.
- Ya que Spring sigue esta forma de securizar las aplicaciones, **sería interesante “repensar” las rutas que le damos a los mappings de los controladores, para agruparlos según su acceso.** Por ejemplo, sería muy cómodo que todas las rutas de acceso público sin registrarse estuviesen bajo una misma ruta, por ejemplo `"/public"`, así podríamos tener `"/public/quienes-somos"`, `"/public/contacta"`, etc. Lo mismo podríamos pensar con tareas de administración, accesibles solo a los administradores de la aplicación, podrían estar bajo `"/admin"`.
- Si una ruta tiene *PathVariable* sería una subruta de la ruta base, por lo que el sistema no distingue `/empl/3` de `/empl/new`. Así pues, tendría sentido lo siguiente:

```
.requestMatchers("/empl/new/**", "/empl/edit/**", "/empl/delete/").hasRole("ADMIN")
.requestMatchers("/empl/**").permitAll()
```

Como evalúa los *requestMatchers* por orden, si la operación es de alta, edición o borrado solo se lo permitirá a los administradores. Para el resto de rutas (esto es, `/empl/` y la variable *PathVariable*) serán de acceso público.

Gestión de usuarios

En el ejemplo anterior, solo teníamos dos usuarios: “user1” y “admin1” creados por *hard-code*. Lo habitual es que las aplicaciones web tengan distintos usuarios y que estos se creen dinámicamente y sean almacenados en una tabla de base de datos.

Para hacer una gestión de usuarios mediante base de datos y no en memoria, lo primero que debemos hacer es decidir qué perfiles o roles de usuarios tendremos en la aplicación y a qué rutas accederán cada uno de ellos. Una vez establecido esto, los pasos a seguir serían:

1.- Debemos tener una clase `SecurityConfig` como vimos en el ejemplo anterior, pero eliminando el bean `UserDetails` ya que es sustituido por el componente que crearemos más adelante y basado en los usuarios de base de datos. Aquí configuraremos mediante los `requestMatchers` los permisos efectivos.

```

@Configuration
@EnableWebSecurity
@EnableMethodSecurity
public class SecurityConfig {
    @Bean
    public AuthenticationManager authenticationManager(
        AuthenticationConfiguration authenticationConfiguration)
        throws Exception {return authenticationConfiguration.getAuthenticationManager();}
    }
    @Bean
    public PasswordEncoder passwordEncoder() { return new BCryptPasswordEncoder(); }

    @Bean
    public SecurityFilterChain filterChain(HttpSecurity http) throws Exception {
        http.headers(
            headersConfigurer -> headersConfigurer
                .frameOptions(HeadersConfigurer.FrameOptionsConfig::sameOrigin));
        http.authorizeHttpRequests(auth -> auth
            .requestMatchers("...").hasAnyRole(...)           //configurar permisos reales
            .requestMatchers(PathRequest.toStaticResources().atCommonLocations())
                .permitAll()                                // para rutas: /css, /js /images
            .anyRequest().authenticated())
            .formLogin(formLogin -> formLogin
                .defaultSuccessUrl("/public/home", true)
                .permitAll())
            .logout(logout -> logout
                .logoutSuccessUrl("/public/home")
                .permitAll())
        // .csrf(csrf -> csrf.disable())
        // .httpBasic(Customizer.withDefaults());
        http.exceptionHandling(exceptions -> exceptions.accessDeniedPage("/accessError"));
        return http.build();
    }
}

```

2.- Debemos guardar en base de datos, para cada usuario, los atributos para la gestión de acceso: serían un nombre de acceso (puede ser el id de la tabla o cualquier otro campo como nombre, o email, etc. siempre que sean únicos para cada usuario), una contraseña (preferiblemente encriptada) y los datos referentes a los permisos que tiene el usuario (lo mínimo sería un único 'rol') aunque podría tener una colección de roles, una colección de permisos individuales, etc.

Para ello podemos crear una clase específica con esos datos o bien ampliar la clase que mantiene los usuarios de la aplicación añadiendo, además de todos los atributos que necesite para el correcto funcionamiento de la aplicación, los atributos que mencionamos en el párrafo anterior. Por simplificar, vamos a optar por esta última opción, aunque hacer una clase específica sería más adecuado.

```

@Entity
public class Usuario {
    @Id
    @GeneratedValue (strategy=GenerationType.IDENTITY)
    private Long id;

    @Column(unique = true)      //evita duplicados a nivel base de datos
    private String nombre;

    private String password;
    private Rol rol;
}

```

Existe una clase llamada 'Role' para gestionar los roles, pero es más sencillo plantearlo como una enumeración y cumple con los requisitos más habituales. Sería un archivo *Rol.java* con este contenido:

```
public enum Rol { ADMIN, TITULAR, USUARIO };
```

3.- Crear un bean que implemente la interfaz *UserDetailsService* que se encarga de la gestión del usuario que se identifica en la aplicación. Esta clase deberá implementar el método *loadUserByUserName(String username)* de la interfaz, que devuelve un objeto de tipo *UserDetails*. En caso de error lanza una excepción *UserNameNotFoundException*. Si tenemos un repositorio JPA para la gestión de usuarios llamado *UserRepository*, podría ser algo así:

```
@Component
public class UserDetailsServiceImpl implements UserDetailsService {
    @Autowired
    private UsuarioRepository usuarioRepository;

    @Override
    public UserDetails loadUserByUsername(String username) throws UsernameNotFoundException {
        Usuario usuario = usuarioRepository.findByNombre(username);
        if (usuario == null) throw (new UsernameNotFoundException("Usuario no encontrado!"));
        return User
            .withUsername(username)
            .roles(usuario.getRol().toString())
            .password(usuario.getPassword())
            .build();
    }
}
```

4.- En el formulario de alta de usuarios habrá que añadir contraseña y uno de los roles predefinidos. Debemos verificar que el nombre es único (o el email o el atributo que empleemos para identificar al usuario en cuanto a seguridad se refiere). Podríamos introducir la contraseña en dos campos de formulario distinto y comprobar que son iguales los dos campos, pero esto sería más una tarea de *frontend* y no de *backend*. También debemos encriptar la contraseña y para ello usaremos el *passwordEncoder* definido en la clase de configuración de seguridad.

```
public Usuario añadir (Usuario usuario) {
    if (usuarioRepository.findByNombre(usuario.getNombre()) != null)
        return null; //ya existe ese nombre de usuario
    String passCrypted = passwordEncoder.encode(usuario.getPassword());
    usuario.setPassword(passCrypted);
    return usuarioRepository.save(usuario);
}
```

El control de duplicados también se puede hacer a nivel de base de datos. Podemos emplear varias formas, la más sencilla sería añadir *@Column(unique=true)* al atributo en cuestión en la entidad *Usuario* y controlando mediante *try...catch* la excepción *DataIntegrityViolationException* que se produciría en caso de introducir un duplicado.

```
public Usuario añadir (Usuario usuario) {
    String passCrypted = passwordEncoder.encode(usuario.getPassword());
    usuario.setPassword(passCrypted);
    try { return usuarioRepository.save(usuario); }
    catch (DataIntegrityViolationException e) {e.printStackTrace(); return null;}
}
```

5.- En la edición de usuario también hay que verificar que, si se cambia el nombre, que el nuevo nombre no esté duplicado tampoco (al igual que en el caso del alta de nuevos usuarios).

```
public Usuario editar(Usuario usuario) {
    String passCrypted = passwordEncoder.encode(usuario.getPassword());
    usuario.setPassword(passCrypted);
    try { return usuarioRepository.save(usuario); }
    catch (DataIntegrityViolationException e) {e.printStackTrace(); return null;}
}
```

6.- En general, el cambio de contraseña puede desligarse del proceso de edición, por lo que sería recomendable hacer un DTO para la edición, que no incluyese la contraseña (y tampoco la fecha de registro, ya que no se debería poder cambiar). Además, como la contraseña está encriptada en la base de datos, en los formularios de edición no se traslada a la vista, como ocurre con el resto de campos.

7.- Para poder iniciar la aplicación, es necesario tener un primer usuario con rol administrador. Lo haríamos en el *CommandLineRunner*.

```
usuarioService.add(new Usuario("admin", "1234", "ADMIN"));
```

8.- Por último, decir que Spring ofrece muchos otros medios de autenticación, por ejemplo, por LDAP.

Obtener el usuario conectado

En muchos casos, para realizar una operación en un servicio o en cualquier otro componente, necesitaremos saber si hay un usuario conectado, y en caso afirmativo saber su nombre o los permisos que tiene. Por ejemplo: en el CRUD de Empleados, imaginemos que solo pudiese modificar los datos de un empleado ese mismo empleado: habría que comprobar si el usuario conectado es igual al usuario que tratamos de modificar. Para ello utilizaremos el objeto *Authentication* que contiene toda la información relativa al usuario conectado:

```
Authentication authentication = SecurityContextHolder.getContext().getAuthentication();
```

Una vez obtenido, podríamos verificar si hay algún usuario conectado y obtener su nombre así:

```
if (!(authentication instanceof AnonymousAuthenticationToken)) {  
    String currentUserName = authentication.getName();  
    // será 'anonymousUser' si no hay usuario logueado
```

Otra información interesante es saber los permisos de dicho usuario. Para ello podemos obtener una colección con todos ellos *authentication.getAuthorities()* que podríamos recorrer para buscar el permiso deseado. Como en nuestro caso, para simplificar, siempre estamos asignando a cada usuario un solo rol: ADMIN, MANAGER, USER, etc... podemos pasar esa colección a String y compararla con ese rol, pero con esta sintaxis: [ROLE_ADMIN], [ROLE_MANAGER], [ROLE_USER], etc.

```
String currentUserRole = authentication.getAuthorities().toString();  
if (currentUserRole.equals("[ROLE_ADMIN]")) { . . . }
```

Estos datos obtenidos se los podríamos pasar al cliente, pero en el caso de aplicaciones Spring MVC con Thymeleaf podemos obtenerlos directamente como veremos en el siguiente apartado.

Mejorando la presentación

Una vez implementado todo el proceso de control de acceso podemos realizar ciertas mejoras como mostrar el nombre del usuario, mostrar ciertas zonas de las vistas solo a determinados roles de usuario (por ejemplo, ciertas opciones de menú), etc.

Thymeleaf posee etiquetas orientadas a la gestión de la seguridad, pero se encuentran en una dependencia adicional, que debemos incorporar en nuestro pom.xml.

```
<dependency>  
    <groupId>org.thymeleaf.extras</groupId>  
    <artifactId>thymeleaf-extras-springsecurity6</artifactId>  
    <version>3.1.2.RELEASE</version>  
</dependency>
```

En la etiqueta inicial <html> incluiremos su referencia:

```
<html xmlns:th="https://www.thymeleaf.org"  
      xmlns:sec="https://www.thymeleaf.org/thymeleaf-extras-springsecurity">
```

Una vez incluida disponemos de estas opciones:

- Para mostrar el nombre de usuario: ``
- Para mostrar una sección solo a usuarios autenticados:
`<div sec:authorize="isAuthenticated()"> Contenido restringido </div>`
- Para mostrar una sección solo a un determinado rol o varios roles:
`<div sec:authorize="hasRole('ADMIN')">Contenido para admin</div>`
`<div sec:authorize="hasAnyRole('ADMIN', 'USER')">Contenido varios roles</div>`

Podríamos hacer el típico menú BootStrap con una opción para login/logout, que muestre también el nombre de usuario registrado y un desplegable para operaciones típicas de usuario: perfil, cambiar contraseña, etc.

En el ejemplo siguiente se muestra ese menú, de forma que si no hay ningún usuario autenticado muestra las opciones del menú superior "Registrarse" e "Iniciar sesión". Por el contrario, si hay algún usuario autenticado, solo se mostraría en el menú superior un elemento con el nombre del usuario, y en el desplegable, las subopciones para: editar el perfil, cambiar la contraseña y cerrar sesión.

```

<nav th:fragment="mainMenu" class="navbar navbar-expand-sm navbar-light bg-light">
  <div class="container-fluid">
    <button class="navbar-toggler" type="button" data-bs-toggle="collapse"
      data-bs-target="#navbarSupportedContent" aria-controls="navbarSupportedContent"
      aria-expanded="false" aria-label="Toggle navigation"><span class="navbar-toggler-icon"></span>
    </button>
    <div class="collapse navbar-collapse" id="navbarSupportedContent">
      <ul class="navbar-nav me-auto mb-2 mb-lg-0">

        <!-- menú común para el público -->
        <li class="nav-item"><a class="nav-link active" aria-current="page" th:href="@{/home/}">
          Inicio</a></li>
        <li class="nav-item"><a class="nav-link active" aria-current="page" th:href="@{/photos/}">
          Fotos</a></li>

        <!-- menú de gestión de usuarios (solo administradores) -->
        <li sec:authorize="hasRole('ADMIN')" class="nav-item">
          <a class="nav-link active" aria-current="page" th:href="@{/usuarios/}">Usuarios</a></li>

        <!-- menú: Registrarse (solo si no autenticado) -->
        <li sec:authorize="!isAuthenticated()">
          <a class="nav-item nav-link active" th:href="@{/registro/nuevo}">Registrarse</a></li>

        <!-- menú: Iniciar sesión (solo si no autenticado) -->
        <li sec:authorize="!isAuthenticated()">
          <a class="nav-item nav-link active" th:href="@{/login}">Iniciar sesión</a></li>

        <!-- menú: nombre-usuario (solo si autenticado) -->
        <li class="nav-item ms-md-auto dropdown" sec:authorize="isAuthenticated()">
          <a class="nav-link dropdown-toggle" href="#" role="button" id="dropdownMenuLink"
            data-bs-toggle="dropdown" aria-haspopup="true" aria-expanded="false">
            <span sec:authentication="name"></span></a>
          <ul class="dropdown-menu dropdown-menu-end" aria-labelledby="dropdownMenuLink">
            <li><a class="dropdown-item" href="#">Editar perfil</a></li>
            <li><a class="dropdown-item" href="#">Cambiar Contraseña</a></li>
            <li><a class="dropdown-item" th:href="@{/logout}">Cerrar sesión GET</a></li>
            <li><form th:action="@{/logout}" method="post">
              <button class="dropdown-item" type="submit">Cerrar sesión POST</button></form></li>
          </ul>
        </li>
      </ul>
    </div></div>
</nav>

```

Nota: se incluyen en el menú anterior las dos opciones para cerrar sesión: logout por 'post' mediante un formulario, que haría el logout directamente, y logout por 'get' mediante un enlace, que nos llevaría a una página intermedia para confirmar el logout o no. Puedes elegir la que quieras.

Login/logout personalizados

Si no queremos usar los formularios de login y logout por defecto, podríamos crear unos propios, teniendo en cuenta que si personalizamos uno debemos personalizar el otro también. El proceso sería el siguiente:

1.- Añadir dos nuevos mappings a un controlador existente o en un nuevo controlador (por ejemplo, *LoginController*). El primero será por GET hacia la ruta del formulario de login ("*/signin*", "*/logir*", etc.) y devolverá una vista con dicho formulario de login, por ejemplo: "*signinView.html*" y el segundo será GET hacia la vista de confirmación de desconexión, también con el nombre que queramos, por ejemplo *signoutView.html*. Los enlaces para conectarse y desconectarse en las vistas apuntarán a estos mappings.

```
@GetMapping("/signin")
public String showLogin() { return "signinView"; }

@GetMapping("/signout")
public String showLogout() { return "signoutView"; }
```

2.- Registrar esas rutas en el bean de configuración, en la sección *formLogin()* y *logout()*:

```
.formLogin(httpSecurityFormLoginConfigurer -> httpSecurityFormLoginConfigurer
    .loginPage("/signin")           // mapping para mostrar formulario de login
    .loginProcessingUrl("/login")   // ruta post de /signin
    .failureUrl("/signin?error")    // vuelve a signin con mensaje de error
    .defaultSuccessUrl("/home", true).permitAll())
    .logout((logout) -> logout
        .logoutSuccessUrl("/signin?logout").permitAll()) // o bien '/home', etc.
```

3.- Crear en */templates/* el archivo *signinView.html* que debe cumplir una serie de restricciones: el formulario debería incluir un 'token csrf' pero Thymeleaf lo hará por nosotros, **no podemos cambiar el atributo name del username y password, y el destino del formulario debe ser por post: /login**. Esta sería una plantilla básica que luego podríamos customizar.

```
<!DOCTYPE html>
<html xmlns="http://www.w3.org/1999/xhtml" xmlns:th="https://www.thymeleaf.org">
<head><title>Autenticación</title></head>
<body><h2>Introduce tus credenciales:</h2>
    <div th:if="${param.error}">Credenciales incorrectas</div>
    <div th:if="${param.logout}">Te has desconectado</div>
    <form th:action="@{/login}" method="post">
        <input type="text" name="username" placeholder="Username"/><br/>
        <input type="password" name="password" placeholder="Password"/><br/>
        <input type="submit" value="Log in" />
    </form>
</body>
</html>
```

Análogamente, para hacer logout, añadiremos un enlace que apunte al *GetMapping ("signout")*. Este mapping servirá una vista que ofrecerá al usuario un formulario para que confirme que desea desconectarse. El submit de este formulario nos llevará a */logout* por POST, que representa la desconexión real. Podría ser algo así:

```
<!DOCTYPE html>
<html lang="es">
    <head><title>Confirmación de desconexión</title>
    </head>
    <body>
        <h2>¿Estás seguro de que quieres desconectarte?</h2>
        <form th:action="@{/logout}" method="post"><input type="submit" value="Logout"/>
    </form>
</body>
</html>
```

Para la operación de `logout` tendríamos otra opción, que sería hacer la desconexión directamente sin pasar por la vista de confirmación. Con `csrf` activado, la ruta `/logout` mediante POST realiza la desconexión efectiva, por lo que, en las vistas, bastaría con añadir el código que hiciese este `post`:

```
<form th:action="@{/logout}" method="post">
    <input type="submit" value="Logout" />
</form>
```

Al pulsar el botón, nos devolvería directamente a la página servida por la ruta especificada en el parámetro `logoutSuccessUrl`.

Resumen para securizar nuestra aplicación

En los puntos anteriores hemos visto todos los aspectos relacionados con la autenticación y autorización en un entorno de aplicación MVC. Vamos a hacer ahora un resumen de los pasos a seguir para implantar todos esos aspectos en un proyecto típico que guarda los usuarios en una tabla de base de datos y que cada uno de ellos dispone de un rol determinado. Serían los siguientes pasos:

- 1.- Incorporar en `pom.xml` las dependencias: `starter-security` y opcionalmente (para emplear las etiquetas Thymeleaf de seguridad): `thymeleaf-extras-springsecurity6`.
- 2.- Crear la clase `SecurityConfig.java` (típicamente en una carpeta `/config /security`) tal cual se muestra unas páginas más atrás y que incluye los beans: `AuthenticationManager`, `PasswordEncoder` y `SecurityFilterChain`.
- 3.- Ajustar en la clase del paso anterior, los permisos de los `mappings` de la aplicación con los `requestMatchers` en función de los roles que tenga la aplicación, en el bean `SecurityFilterChain`.
- 4.- Crear la enumeración ROL. Crear la entidad Usuario, que incluya entre sus atributos el rol asignado, además de un atributo para login (puede ser nombre, dni, email, etc.) y contraseña.
- 5.- Crear la clase `UserDetailsServiceImp` tal cual se muestra en las páginas anteriores.
- 6.- Modificar el CRUD de usuario para que, al añadir y editar, encripte la contraseña y que verifique que el campo que usemos para login (nombre, email, etc.) no tenga duplicados.
- 7.- Opcionalmente, crear vistas personalizadas para login y logout, con los correspondientes controladores y la modificación en el `SecurityFilterChain`.
- 8.- Si deseamos incluir en la lógica de negocio criterios referentes al usuario conectado y/o a su rol, emplearemos el objeto `Authentication`.
- 9.- Si deseamos incluir el usuario y/o rol en las vistas: emplearemos el atributo: `sec:authorize` y añadiremos a la etiqueta `<html>` el atributo `xmlns:sec`.

Otras opciones sobre autenticación

Autoregistro de usuarios

Para que un usuario no identificado pueda registrarse, deberíamos crear una nueva entrada en el menú principal, un nuevo controlador y vistas, todo ello similar a la gestión de usuarios, pero más limitados, ya que el rol asignado será el más básico, no lo podrá seleccionar el usuario. Debemos dar permisos a todo el mundo a acceder a esta nueva URL.

```
@Bean
public SecurityFilterChain filterChain(HttpSecurity http) throws Exception {
    http.authorizeHttpRequests(auth -> auth
        .requestMatchers ("/registro/nuevo/**").permitAll()
```

Por otra parte, en muchos casos será aconsejable que el usuario proporcione una dirección de correo electrónico y que se valide el usuario cuando se envíe un email de confirmación. Dejamos esta parte fuera de este manual, aunque el proceso sería el siguiente:

1.- Añadir a la clase Usuario la dirección de correo electrónico y un nuevo campo, de tipo boolean, llamado "activo" o similar. Cuando el usuario se registrase estaría a false y no podría loguearse.

2.- En el proceso de alta se enviaría a esa dirección un email para que validase el usuario. Tendríamos un controlador que recibiese el enlace enviado al correo y que pasase a true el campo "activo".

Otra opción sería tener una tabla de usuarios no confirmados donde se almacenarían en el proceso de autoregistro. Una vez recibido el email de confirmación, el usuario pasaría a la tabla "real" de usuarios. Así evitamos tener en cuenta en todas las operaciones ese campo boolean "activo" del que hablamos en el primer punto.

Opción "Recuérdame"

Cada vez que un usuario visita nuestra aplicación que requiere autentificación debe iniciar de nuevo sesión y esto puede llegar a ser tedioso. La funcionalidad "Recuérdame" es un mecanismo para resolver esto de forma que si, una vez identificados, abandonamos el sitio sin cerrar sesión, nuestras credenciales serán recordadas por cierto tiempo, y así al volver a visitar el sitio, no necesitaremos identificarnos de nuevo. Obviamente, este mecanismo tiene problemas de seguridad en caso de alguien ajeno acceda a nuestro navegador.

Spring ofrece dos vías de implementarlo: mediante cookies o mediante persistencia. En este manual vamos a ver solo la primera.

Al loguearnos normalmente se almacena una cookie llamada `JSESSIONID` con los datos de nuestra sesión. Esta cookie expira al cerrar el navegador. De todas formas, cuando cerramos sesión en la aplicación, esta cookie ya no funcionará por lo que no es necesario borrarla explícitamente.

Con la opción "Recuérdame" adicionalmente se almacenará una nueva cookie que tiene una vigencia superior a la de la sesión (por defecto dos semanas) que nos permitirá seguir logueados sin necesidad de autentificarnos de nuevo. Esta cookie obviamente guarda el usuario y contraseña, por lo que puede ser peligroso si es interceptada.

Lo único que debemos hacer es añadir al bean SecurityFilterChain la opción `rememberMe`:

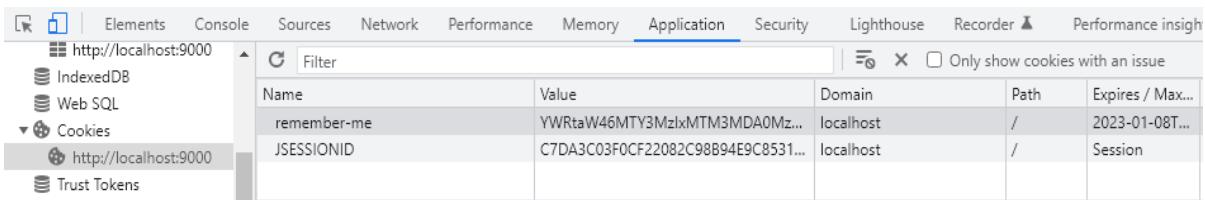
```
@Bean
public SecurityFilterChain filterChain(HttpSecurity http) throws Exception {
    http.headers()
        .headersConfigurer -> headersConfigurer
            .frameOptions(HeadersConfigurer.FrameOptionsConfig::sameOrigin);
    http.authorizeHttpRequests(auth -> auth
        .requestMatchers("...").hasAnyRole("..."))
        .rememberMe(Customizer.withDefaults())
        .httpBasic(Customizer.withDefaults());
    http.exceptionHandling(exceptions -> exceptions.accessDeniedPage("/accessError"));
    return http.build();
}
```

Con solo esta operación se hará toda la gestión de la cookie que recordará nuestro usuario y contraseña logueado. También modifica automáticamente el formulario de login por defecto, añadiéndole el checkbox que debe marcar el usuario si quiere que guarde sus credenciales.

Si hemos desarrollado un formulario de login personalizado debemos añadirle nosotros mismos este checkbox, y su propiedad `name` debe ser tal y como se muestra a continuación:

```
<input type='checkbox' name='remember-me' />Remember me on this computer</p>
```

Para ver las cookies desde Google Chrome, accedemos a las herramientas para desarrolladores (Cntrl + May + I) y en la pestaña *Aplicación*, en el menú lateral: *Storage > Cookies*:



Name	Value	Domain	Path	Expires / Max...
remember-me	YWRtaW46MTY3MzIxMTM3MDA0Mz...	localhost	/	2023-01-08T...
JSESSIONID	C7DA3C03F0CF22082C98B94E9C8531...	localhost	/	Session

Olvidé mi contraseña

Otro aspecto que puede ser interesante incorporar a nuestro sistema es una opción para restablecer la contraseña en caso de olvido. La forma más habitual de hacer esto es enviando un email a usuario con un enlace para establecer la nueva contraseña. Obviamente, debemos incorporar este nuevo dato, la dirección de email, en el proceso de registro de usuarios.

No lo vamos a desarrollar, pero en estos enlaces tenéis el proceso detallado:

<https://stackabuse.com/spring-security-forgot-password-functionality/>

<https://www.baeldung.com/spring-security-registration-i-forgot-my-password#:~:text=Request%20the%20Reset%20of%20Your,an%20email%20to%20the%20user.>

Tema 9: API Rest

Introducción

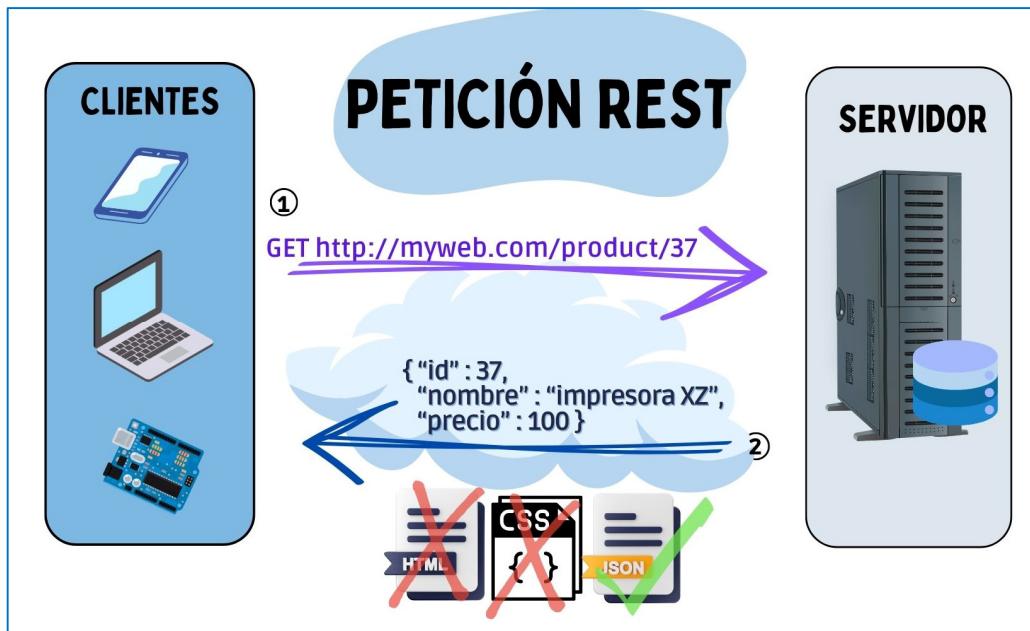
Las aplicaciones realizadas hasta ahora incluyen en un solo proyecto todas las capas necesarias: acceso a los datos, lógica de negocio, presentación, etc. de forma que no hay elementos externos (salvo navegador de usuario y gestor de base de datos).

Esta arquitectura tiene ventajas como la uniformidad de desarrollo (todo en el mismo lenguaje), el despliegue es sencillo (un solo jar o war) y es más fácil a la hora de aprender una tecnología y también para proyectos pequeños.

Como inconveniente fundamental **de estas aplicaciones monolíticas podemos decir que no responden ante clientes heterogéneos**: una aplicación en Angular, una aplicación móvil, dispositivos IoT, etc. Nuestra aplicación solo responderá al cliente para el que fue diseñada (en nuestro caso, el navegador web). Además, el escalado de este tipo de aplicaciones es más complejo: no sería sencillo balancear distintas partes de la aplicación en distintos servidores ya que la aplicación está muy cohesionada (*por ejemplo, en una aplicación de videos online, sería aconsejable asignarle más recursos a servir videos a los usuarios que no para el proceso de login*).

La solución a estos problemas son los servicios distribuidos que tratan de desacoplar algunos elementos de nuestra aplicación, por ejemplo, la capa de presentación, la lógica de negocio, etc. Ha habido muchas tecnologías a lo largo del tiempo para el desarrollo de este tipo de arquitectura: *CORBA, RMI, luego SOAP, siendo gRPC y REST las más empleadas actualmente*.

Con REST lo que haremos es que nuestra aplicación no tenga capa de presentación, sino que ofrecerá sus servicios a cualquier tipo de cliente, que nos harán peticiones y nosotros simplemente le serviremos los datos solicitados.



Descripción de API Rest

REST es un nuevo enfoque propuesto por Roy Fielding en su tesis doctoral y estas son sus bases:

- La comunicación entre el cliente y el servidor se hace mediante el protocolo HTTP.
- Es una comunicación sin estados. Cada petición-respuesta es completa, no necesita sincronizarse con otras peticiones.
- Los recursos a los que accede el cliente se mapean mediante una URL y se denominan **end points**.

- Los datos se envían al cliente, normalmente en formato JSON. El cliente lo procesará como desee.
- No es propio de Java, se puede implementar con cualquier lenguaje.
- El cliente no tiene por qué ser solo un cliente final, puede ser otra aplicación. Pensemos en una aplicación que necesita en un momento dado un dato meteorológico, una cotización en bolsa, etc. Puede la aplicación hacer una solicitud REST a otro servidor que ofrezca tal servicio.
- Son la base para la arquitectura de aplicaciones en microservicios, que veremos más adelante.

Al estar basado en HTTP es buen momento para volver al Tema 1 y repasar los conceptos que aquí necesitaremos como son: estructura de la petición, estructura de la respuesta, verbos (GET, POST, PUT, etc.), códigos de respuesta, etc.

Los verbos HTTP que más usaremos:

- **GET**: solicitar un recurso al servidor, por ejemplo, para **consultas**.
- **POST**: enviar datos en el cuerpo de la petición para **crear** nuevos recursos en la aplicación.
- **PUT**: enviar datos en el cuerpo de la petición para **editar** recursos en la aplicación. Algunos desarrolladores usan POST para esta misión, pero no es correcto.
- **DELETE**: **borrar** un recurso del servidor.



API Rest en Spring

Spring a través de su dependencia **starter-web** nos facilita el trabajo de "restificar" nuestras aplicaciones de forma sencilla. Estas son sus características:

- El cambio con respecto a las aplicaciones anteriores solo afecta a la capa del controlador y vistas, **manteniendo sin cambios los servicios, repositorios, modelo de dominio, etc.**
- Sustituimos la anotación `@Controller` por **`@RestController`** (Controller + `ResponseBody`) que hace que sus métodos por defecto devuelvan un cuerpo de respuesta HTTP. *Recordemos que `@Controller` devolvía un `String` que representaba una vista.*
- Las vistas, archivos `html` estáticos y `css` no tienen sentido en este tipo de aplicaciones. Serán las aplicaciones clientes de los distintos dispositivos las que gestionen todos estos recursos. El servidor solo sirve los datos.
- Spring realiza la conversión de clase Java a cuerpo de respuesta, así nuestro método de controlador puede devolver una clase de nuestro modelo o un DTO, que se convertirá de forma transparente para nosotros en un objeto para el peticonario, normalmente en formato JSON.

Ejemplo: Suponiendo que tenemos una clase `Empleado` (id, nombre, email, salario) con su repositorio JPA, así sería un `RestController` para devolver un archivo JSON con los datos de un empleado.

```
@RestController
public class EjemploController {

    @Autowired private RepositorioEmpleado repositorioEmpleado;

    @GetMapping("/buscar/{id}")
    public Empleado getEmpleado(@PathVariable Long id) {
        Empleado empleado = repositorioEmpleado.findById(id).orElse(null);
        return empleado;
    }
}
```

Nos puede llamar la atención que devuelve un objeto empleado, no una respuesta HTTP con código de estado, cuerpo, etc. Lo explicaremos más adelante.

- Disponemos de distintas clases para trabajar con las peticiones y respuestas HTTP como son:
 - **HttpMessageConverter**: se encarga de la conversión de clases a JSON y viceversa usando las librerías Jackson.
 - **HttpEntity<T>** y sus dos subclases **RequestEntity<T>** y **ResponseEntity<T>**: representan una petición o respuesta HTTP completa con su cabecera y cuerpo. En el ejemplo anterior estamos devolviendo un empleado en el cuerpo de la respuesta, pero no gestionamos otros parámetros como el código de respuesta **HttpStatus**, lo hace Spring, pero con valores por defecto. Si empleamos estas clases tendremos un control más detallado.
 - **HttpHeaders**: representa los encabezados de una petición o de una respuesta.
 - **RestClient** (antes *WebClient* y *RestTemplate*): se emplea si queremos que nuestra API sea a su vez cliente y haga peticiones a otras API Rest remotas.
- **Spring Data Rest**: Permite transformar un repositorio de Spring Data en API Rest de forma sencilla sin apenas añadir código alguno.
- **Spring HATEOAS**: (Hypermedia as Engine of Application State) permite incluir enlaces en los resultados devueltos en una respuesta REST para que el cliente pueda navegar de unos recursos a otros.

ResponseType <T>

Es una clase que nos va a permitir manejar la respuesta que damos a nuestros clientes de una forma más conveniente. Es una clase hija de **HttpEntity<T>** que añade un atributo más para el código de estado de la respuesta.

En el apartado anterior vimos como en un método del RestController podíamos devolver una clase de nuestro modelo y que Spring se encargaba de convertirlo al cuerpo de una respuesta HTTP. Lo que no dijimos es que la cabecera de esa respuesta se construía por defecto, y no teníamos forma de customizarla. Si lo que devolvemos es un **ResponseType** podremos configurar todos los parámetros de esa cabecera (sobre todo el código de estado: 200, 201, 404, etc.)

El proceso a seguir sería hacer que los métodos del RestController, en vez de devolver una clase del modelo, devuelvan un objeto de **ResponseType**. La clase dispone de diversos métodos estáticos que nos permiten construir la respuesta de forma sencilla. El método principal sería:

```
ResponseType.status(n).body(recurso);
```

siendo '*n*' un código de estado, 200, 201... y '*recurso*' el recurso que queremos enviar. Para los códigos tenemos la enumeración **HttpStatus** con valores OK, CREATED, NO_CONTENT, NOT_FOUND, FORBIDDEN, BAD_REQUEST, etc. En caso de que el cuerpo de la respuesta vaya vacío (típico por ejemplo en borrados), sustituiríamos **.body(recurso)** por **.build()**. Ejemplos:

```
ResponseType.status(HttpStatus.CREATED).body(empleado);
ResponseType.status(HttpStatus.NO_CONTENT).build();
```

Los códigos de respuesta habituales para cada operación son:

- **Get**: 200 (OK) si localizamos el recurso, 404 (NOT_FOUND) si no lo localizamos.
- **Post**: 201 (CREATED) si alta ok. 400 (BAD_REQUEST) si los datos recibidos son erróneos.
- **Put**: 200 (OK) si modificamos el recurso, 404 (NOT_FOUND) si no encuentra el recurso a modificar, y 400 (BAD_REQUEST) si los datos recibidos son erróneos.
- **Delete**: 204 (NO_CONTENT) y sin cuerpo si encuentra el elemento a borrar o 404 (NOT_FOUND) si no encuentra el recurso a borrar.

ResponseEntity dispone de métodos adicionales, que agrupan el código de respuesta y el estado, pero no serían necesario, simplemente nos permiten escribir la respuesta de una forma abreviada:

`ResponseEntity.ok(e)` siendo 'e' una instancia de la clase que queremos enviar: Empleado, Producto, etc, devuelve una respuesta con estado 200 y la clase en el cuerpo.

`ResponseEntity.notFound().build()`; Construye una respuesta con código 404 y cuerpo vacío.

`ResponseEntity.noContent().build()`; Construye una respuesta con código 204 y cuerpo vacío, típica de operaciones de borrado.

Por ejemplo, para una inserción directamente en el repositorio, sin servicio intermedio, ni controlar posibles errores:

```
@PostMapping ("/nuevo")
public ResponseEntity<Empleado> nuevoEmpleado (@RequestBody Empleado nuevo){
    Empleado guardado = empleadoRepositorio.save (nuevo);
    return ResponseEntity.status(HttpStatus.CREATED).body(guardado);
}
```

En este ejemplo hemos usado la anotación `@RequestBody`. Esta anotación será fundamental en las peticiones que lleven datos asociados ya que permite injectar en el método los datos recibidos, probablemente JSON, pero para ser tratados en el método como cualquier otra clase Java. Los casos típicos de su uso en un CRUD son en el alta y modificación de recursos, no así en consultas y borrados en las que no recibimos datos adicionales.

Cuando el elemento que añadimos al cuerpo es una entidad de nuestro modelo de dominio, éste se transforma a formato JSON automáticamente. Cuando lo que enviamos es un valor sencillo (por ejemplo, un entero, o un par de enteros) lo habitual es definir un DTO con la estructura de esa respuesta, bien como clase:

```
@Data
@AllArgsConstructor
public class MyDto {
    private String nombre;
    private Integer edad;
}
```

bien como record:

```
public record MyDto (String nombre, Integer edad) {}
```

Y devolver al cliente una instancia del dto:

```
return ResponseEntity.ok(new MyDto("Pepe", 20));
```

También podríamos prescindir del DTO, y construir un mapa (la clave de cada elemento del mapa sería una cadena con el nombre del atributo y el valor de cada elemento del mapa sería el valor del atributo que llegaría al usuario).

```
String nombre = "Pepe";
Integer edad = 20;
Map<String, String> mapa = new LinkedHashMap<>();
mapa.put("nombre", nombre);
mapa.put("edad", edad.toString());
return ResponseEntity.ok(mapa);
```

Proyecto API Rest

Partiremos del ejemplo del CRUD de empleado del tema 7 para convertirla en un servicio REST. Los pasos a seguir serían para "restificar" el proyecto de empleado serían:

1. Eliminar del pom.xml la dependencia `Thymeleaf`, ya no es necesaria.
2. Eliminar las vistas de la carpeta `templates`

3. Sustituir la anotación `@Controller` por `@RestController` en el controlador.
4. En el método de actualización de un recurso, sustituir la anotación `@PostMapping` por `@PutMapping`, y en el método de eliminación sustituir `@GetMapping` por `@DeleteMapping`. Como no hay formulario de entrada de datos ya no es necesario el `@GetMapping` que mostraba el formulario.
5. En el caso de añadir un nuevo empleado y actualizar empleado necesitamos extraer los datos que enviará el cliente al servidor, lo haremos mediante la anotación `@RequestBody` que comentamos en el apartado anterior. Podemos seguir utilizando `@Valid` para validar las anotaciones de validación de los datos recibidos desde el cliente (`@Min`, `@Email`, `@NotEmpty`, etc.) pero no es necesario incluir `BindingResult`, en caso de que no se cumplan las restricciones devolverá código 400 BAD REQUEST. En el próximo apartado de "Gestión de errores" explicaremos en detalle este aspecto.

```
@PutMapping("/empleado/{id}")
public ResponseEntity<?> editElement(@Valid @RequestBody Empleado editEmpleado ){
```

6. La respuesta será un `ResponseEntity` gestionando los códigos de respuesta de las peticiones y el cuerpo de la respuesta, como hemos descrito previamente:
 - `@GetMapping: 200 (OK)` si localizamos el recurso, `404 (Not found)` si no lo localizamos.
 - `@PostMapping: 201 (Created)` si alta ok. `400 (Bad request)` si los datos recibidos son erróneos.
 - `@PutMapping: 200 (OK)` si modificamos el recurso, `404 (Not found)` si no encuentra el recurso a modificar, y `400 (Bad request)` si los datos recibidos son erróneos.
 - `@DeleteMapping: 204 (No Content)` y sin cuerpo si encuentra el elemento a borrar o `404 (Not found)` si no encuentra el recurso a borrar.
7. En MVC empleábamos solo Get y Post, por lo que debíamos diferenciar las URL para cada operación: /new, /edit, /delete, etc. Ahora como cada operación CRUD va a tener su mapping: `@GetMapping`, `@PostMapping`, `@PutMapping`, `@DeleteMapping`, podemos usar la misma URL para todas las operaciones.

Vamos a ver el código final resultante, por ahora sin tratamiento de errores y suponiendo que el servicio inyectado devuelve `null` en caso de error. En el siguiente apartado refinaremos este código para que los métodos de servicio lancen excepciones en vez de devolver `null` y también que el controlador sea capaz de gestionar esas excepciones. Así tendremos un código mucho más limpio, pero por ahora sería así:

```
@RestController
public class EmpleadoController {
    @Autowired
    private EmpleadoService empleadoService;

    @GetMapping("/empleado")
    public ResponseEntity<?> getList() {
        List<Empleado> listaEmpleados = empleadoService.obtenerTodos();
        if (listaEmpleados.isEmpty())
            return ResponseEntity.notFound().build(); // cod 404
        else
            return ResponseEntity.ok(listaEmpleados); // cod 200
        //return ResponseEntity.status(HttpStatus.OK).body(ListaEmpleados);
    }

    @GetMapping("/empleado/{id}")
    public ResponseEntity<?> getOneElement(@PathVariable Long id) {
        Empleado empleado = empleadoService.obtenerPorId(id);
        if (empleado == null)
            return ResponseEntity.notFound().build(); // cod 404
        else
            return ResponseEntity.ok(empleado); // cod 200
    }
}
```

```

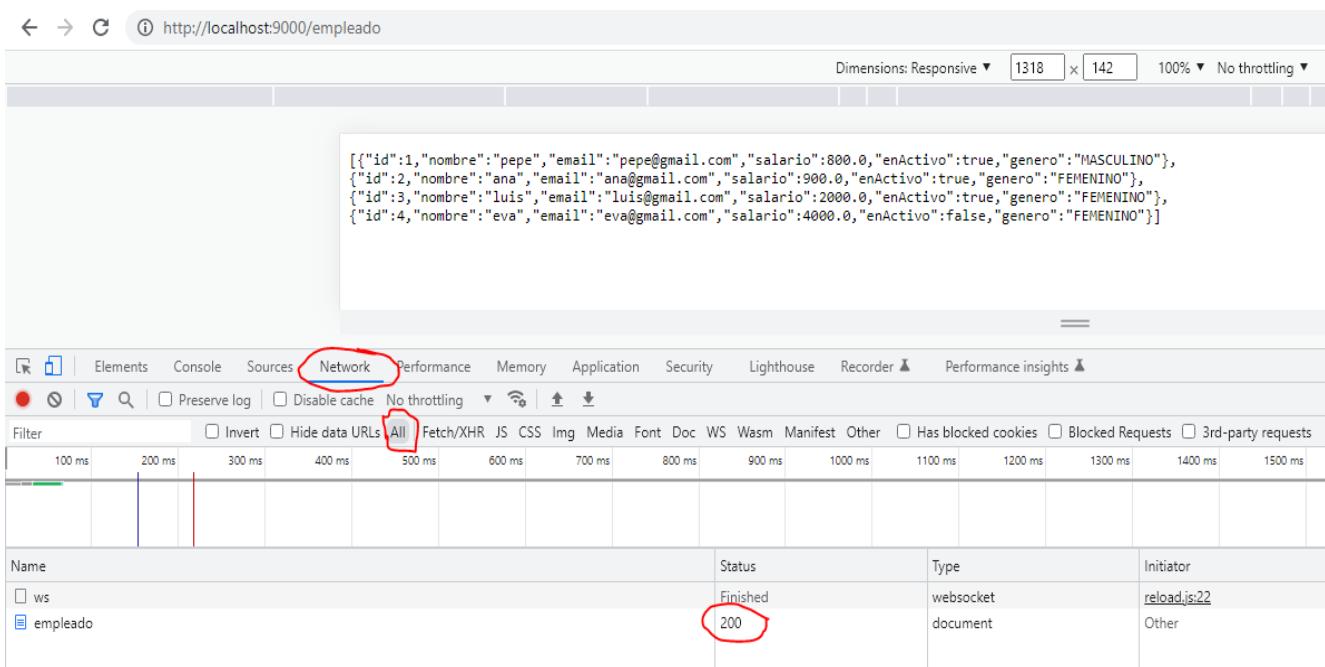
@PostMapping("/empleado")
public ResponseEntity<?> newElement(@Valid @RequestBody Empleado nuevoEmpleado) {
    // @Valid si no se cumple la validación devuelve BAD_REQUEST cod 400
    Empleado empleado = empleadoService.añadir(nuevoEmpleado);
    return ResponseEntity.status(HttpStatus.CREATED).body(empleado); // cod 201
}

@PutMapping("/empleado/{id}")
public ResponseEntity<?> editElement(@Valid @RequestBody Empleado editEmpleado,
                                         @PathVariable Long id) {
    // @Valid si no se cumple la validación devuelve BAD_REQUEST cod 400
    Empleado empleado = empleadoService.obtenerPorId(id);
    if (empleado == null)
        return ResponseEntity.notFound().build(); // cod 404
    else {
        empleado = empleadoService.editar(editEmpleado);
        return ResponseEntity.ok(empleado); // cod 200
    }
}
@DeleteMapping("/empleado/{id}")
public ResponseEntity<?> deleteElement(@PathVariable Long id) {
    Empleado empleado = empleadoService.obtenerPorId(id);
    if (empleado == null)
        return ResponseEntity.notFound().build(); // cod 404
    empleadoService.borrar(id);
    return ResponseEntity.noContent().build(); // cod 204
}
}

```

Probar con Postman

Con este tipo de aplicaciones no tenemos vistas con enlaces o botones para navegar por nuestra aplicación. Las peticiones GET podríamos probarlas desde el propio navegador. Si hacemos la petición de un recurso, podemos seleccionar: *Inspeccionar > Network* (pestaña All) y ver la respuesta recibida.



The screenshot shows the Network tab of the Chrome DevTools. The URL in the address bar is `http://localhost:9000/empleado`. The response body is a JSON array of employees:

```

[{"id":1,"nombre":"pepe","email":"pepe@gmail.com","salario":800.0,"enActivo":true,"genero":"MASCULINO"}, {"id":2,"nombre":"ana","email":"ana@gmail.com","salario":900.0,"enActivo":true,"genero":"FEMENINO"}, {"id":3,"nombre":"luis","email":"luis@gmail.com","salario":2000.0,"enActivo":true,"genero":"FEMENINO"}, {"id":4,"nombre":"eva","email":"eva@gmail.com","salario":4000.0,"enActivo":false,"genero":"FEMENINO"}]

```

The status column for the request shows '200'.

Para otras peticiones como PUT y POST esto ya no es posible, debemos usar herramientas para enviar peticiones con cuerpo. Herramientas típicas para esta tarea son: **CURL** (desde línea de comandos, más tediosa de utilizar), **Postman** (aplicación gráfica muy potente) y si trabajamos con Visual Studio Code, disponemos también de la extensión **Thunder Client** con un comportamiento similar a Postman.

Vamos a emplear Postman: la descargamos desde: <https://www.postman.com/downloads/>. En el proceso de instalación solicita si queremos crear una cuenta, seleccionamos "skip and go to the app" y podemos trabajar sin registrarnos. La forma de trabajar es muy sencilla:

- Abrir una nueva pestaña en la pantalla principal de la aplicación.
- Seleccionar el verbo HTTP en el desplegable.
- En caso de que la petición lleve consigo datos, seleccionamos la pestaña "Body", tipo "JSON" y añadimos los datos necesarios.
- Pulsamos el botón *[Send]* y en la parte de abajo veremos la respuesta (cabecera con código de respuesta y body).

The screenshot shows the Postman interface with a POST request to `localhost:9000/empleado/`. The **Body** tab is selected, showing a JSON payload:

```

1
2   ...
3   ...
4   ...
5   ...
6   ...
7   ...
8
  
```

The response section shows a **201 Created** status with a **21 ms** response time and **269 B** size. The response body is identical to the request body.

Postman nos permite almacenar todas las peticiones que realizamos. Las agrupa en "Colecciones" que son simplemente grupos de peticiones. Podemos crear una colección por cada proyecto. Cuando tengamos una petición lista y veamos que nos devuelve los resultados deseados, en el botón **Save as...** podemos añadirla a la colección actual.

The screenshot shows the Postman interface with the **Scratch Pad** selected. A collection named `proy0801` is expanded, showing a **POST** request to `localhost:9000/empleado/`. The request body is identical to the one in the previous screenshot.

Importante: si quieras probar todo esto desde una aplicación del lado de cliente, por ejemplo, desarrollada en JavaScript vete al apartado “Consumiendo APIs” y añade la gestión de **CORS** al proyecto en Spring, de lo contrario no funcionará tu aplicación cliente.

Añadir DTO al proyecto

Ya comentamos en el tema anterior la necesidad de los DTO para evitar el envío de datos innecesarios al cliente. En el ejemplo que estamos tratando, quizás para el listado de todos los empleados, con mandar el nombre y el email sea suficiente, y solo cuando se piden los datos de uno concreto ya informar del resto atributos.

En el caso de aplicaciones REST se presenta un problema adicional: cuando tenemos varias relaciones entre nuestras entidades el envío de los objetos de dominio puede reducir la claridad de los datos que se transfieren. Volvamos a la asociación vista en el tema anterior en la que Empleado estaba vinculado con su Departamento con una asociación @ManyToOne. Si no empleamos DTO, ante un “get” devolveríamos algo así, quizás más difícil de procesar por el cliente:

```

1
2     "id": 2,
3     "nombre": "ana",
4     "email": "ana@gmail.com",
5     "salario": 900.0,
6     "enActivo": true,
7     "genero": "FEMENINO",
8     "departamento": {
9         "id": 2,
10        "nombre": "Comercial"
11    }
12

```

Y cuando el cliente quisiese editar un empleado con “put” debería enviar el mismo formato, y en caso de un alta de empleado, algo en un formato similar (igual, pero sin atributo id). Si pensamos en que el empleado tenga más asociaciones, vemos la complejidad de los datos a transferir.

Vamos emplear DTOs para resolver las dos situaciones comentadas: el envío de demasiados datos para el listado de empleados y conseguir un formato más simple para la asociación con Departamento. Estos serían los pasos a seguir:

1.- Añadimos al pom.xml del proyecto la dependencia necesaria para emplear ModelMapper y creamos el archivo de configuración con la creación del bean de ModelMapper, según se indicaba en el capítulo anterior.

2.- Añadimos al proyecto la entidad Departamento con repositorio, servicio y controlador (@RestController). Podríamos tomarlos del ejemplo del tema anterior en el que se plasmaba esta asociación y “restificar” el controlador de Departamento.

```

@RestController
public class DepartamentoController {
    @Autowired
    private DepartamentoService departamentoService;

    @GetMapping("/depto")
    public ResponseEntity<?> getList() {
        List<Departamento> listaDepartamento = departamentoService.obtenerTodos();
        if (listaDepartamento.isEmpty())
            return ResponseEntity.notFound().build();
        else
            return ResponseEntity.ok(listaDepartamento);
    }
    ...
}

```

3.- Creamos una carpeta llamada "dto" (por ejemplo, debajo de la carpeta "domain" o donde queramos) y añadimos dos DTO, uno para usar cuando se solicite el listado de todos los empleados (solo con id, nombre e email) y otro para cuando se crea un nuevo empleado con "post" para tener esa estructura más simple. Este último lo podremos usar también para las ediciones de empleados "put".

```

@Getter
@Setter
public class EmpleadoDto {
    private Long id;
    private String nombre;
    private String email;
}

@Getter
@Setter
public class EmpleadoNuevoDto {
    private String nombre;
    private String email;
    private Double salario;
    private boolean enActivo;
    private Genero genero;
    private Long departamentoId;
}

```

4.- En el controlador de Empleado inyectamos el ModelMapper.

```

@Autowired
private ModelMapper modelMapper;

```

5.- En el controlador de Empleado, para el "get" de todos los empleados empleamos el primer DTO:

```

@GetMapping("/empleado")
public ResponseEntity<?> getList() {
    List<Empleado> listaEmpleados = empleadoService.obtenerTodos();
    if (listaEmpleados.isEmpty())
        return ResponseEntity.notFound().build(); // cod 404
    else {
        List<EmpleadoDto> listaEmpleadoDto = new ArrayList<>();
        for (Empleado e : listaEmpleados)
            listaEmpleadoDto.add(modelMapper.map(e, EmpleadoDto.class));
        return ResponseEntity.ok(listaEmpleadoDto); // cod 200
    }
}

```

6.- En el controlador de Empleado, para el "post" emplearemos el segundo DTO y debemos convertirlo de ese DTO a una instancia de Empleado para añadirlo al repositorio. Vamos a hacerlo primero de una forma más tosca, sobre el constructor en el mismo controlador (luego lo mejoraremos).

```

@PostMapping("/empleado")
public ResponseEntity<?> newElement(@RequestBody EmpleadoNuevoDto empleadoNuevoDto){
    Empleado empleado = new Empleado(null, //id autogenerado
        empleadoNuevoDto.getNombre(),
        empleadoNuevoDto.getEmail(),
        empleadoNuevoDto.getSalario(),
        empleadoNuevoDto.isEnActivo(),
        empleadoNuevoDto.getGenero(),
        departamentoService.obtenerPorId(empleadoNuevoDto.getDepartamentoId()));

    Empleado empleadoSaved = empleadoService.añadir(empleado);
    return ResponseEntity.status(HttpStatus.CREATED).body(empleadoSaved); // cod 201
}

```

7.- Y hacemos lo mismo para el “put”, aunque en este caso en el constructor de empleado sí incluye el id de empleado, ya que no es un alta, es una edición de un empleado existente:

```

@PutMapping("/empleado/{id}")
public ResponseEntity<?> editElement(@RequestBody EmpleadoNuevoDto editEmpleado,
    @PathVariable Long id) {
    Empleado empleado = empleadoService.obtenerPorId(id);

    if (empleado == null)
        return ResponseEntity.notFound().build();
    else {
        empleado = new Empleado(
            id,
            editEmpleado.getNombre(),
            editEmpleado.getEmail(),
            editEmpleado.getSalario(),
            editEmpleado.isEnActivo(),
            editEmpleado.getGenero(),
            departamentoService.obtenerPorId(editEmpleado.getDepartamentoId()));
        Empleado empleadoSaved = empleadoService.editar(empleado);
        return ResponseEntity.ok(empleadoSaved);
    }
}

```

8.- Estas conversiones en el controlador se podrían pasar a una clase que se encargase de esta tarea. Podemos situarla en la carpeta/paquete “services”, ya que es lógica de negocio (*o bien en otra carpeta a la que podríamos llamar por ejemplo “utilities”*). También podemos meterla en el servicio.

```

@Component
public class EmpleadoDtoConverter {

    @Autowired private DepartamentoService departamentoService;

    @Autowired private ModelMapper modelMapper;

    public Empleado convertDtoToEmpleado(EmpleadoNuevoDto empleadoNuevoDto) {
        return new Empleado(null,
            empleadoNuevoDto.getNombre(),
            empleadoNuevoDto.getEmail(),
            empleadoNuevoDto.getSalario(),
            empleadoNuevoDto.isEnActivo(),
            empleadoNuevoDto.getGenero(),
            departamentoService.obtenerPorId(empleadoNuevoDto.getDepartamentoId()));
    }

    public Empleado convertDtoToEmpleado(EmpleadoNuevoDto empleadoEditDto, Long id) {
        Empleado empleado = convertDtoToEmpleado(empleadoEditDto);
        empleado.setId(id);
        return empleado;
    }

    public EmpleadoDto convertEmpleadoToDto(Empleado empleado) {
        return modelMapper.map(empleado, EmpleadoDto.class);
    }
}

```

Así tendríamos unos métodos en el controlador más elegantes (inyectando esta nueva clase):

```

@RestController
public class EmpleadoController {

    @Autowired
    private EmpleadoDtoConverter empleadoDtoConverter;

    ...

```

```

@GetMapping("/empleado")
public ResponseEntity<?> getList() {
    List<Empleado> listaEmpleados = empleadoService.findAll();
    if (listaEmpleados.isEmpty()) return ResponseEntity.notFound().build();
    else {
        List<EmpleadoDto> listaEmpleadoDto = new ArrayList<>();
        for (Empleado empleado : listaEmpleados)
            listaEmpleadoDto.add(empleadoDtoConverter.convertEmpleadoDto(empleado));
        return ResponseEntity.ok(listaEmpleadoDto);
    }
}

@PostMapping("/empleado")
public ResponseEntity<?> newElement(@RequestBody EmpleadoNuevoDto empleadoNuevoDto) {
    Empleado empleado = empleadoDtoConverter.convertDtoToEmpleado(empleadoNuevoDto);
    Empleado empleadoSaved = empleadoService.agregar(empleado);
    return ResponseEntity.status(HttpStatus.CREATED).body(empleadoSaved);
}

@PutMapping("/empleado/{id}")
public ResponseEntity<?> editElement(@RequestBody EmpleadoNuevoDto editEmpleado,
    @PathVariable Long id) {

    Empleado empleado = empleadoService.obtenerPorId(id);
    if (empleado == null)
        return ResponseEntity.notFound().build(); // cod 404
    else {
        empleado = empleadoDtoConverter.convertDtoToEmpleado(editEmpleado, id);
        Empleado empleadoSaved = empleadoService.editar(empleado);
        return ResponseEntity.ok(empleadoSaved); // cod 200
    }
}

```

Relaciones bidireccionales

Cuando tenemos relaciones bidireccionales entre entidades (@OneToMany y @ManyToOne) por ejemplo como hemos visto en ejemplos y ejercicios anteriores entre Empleado y Categoría se pueden producir bucles infinitos a la hora de recuperar los datos de una petición: un empleado llevaría asociado una categoría, y esa categoría sus empleados, y así sucesivamente, cada uno de sus empleados su categoría, etc.

Para evitar este problema de recursividad tenemos varias opciones, una de ellas es generar clases DTO que no contengan la relación y adaptar controladores y servicios para trabajar con estos DTO allí donde sea necesario.

Otra opción, más sencilla, es incluir la etiqueta `@JsonIgnore` en la clase “1” de la relación “1 a n”, para el atributo con la lista de elementos “n”. Así a la hora de generar la respuesta en formato JSON para enviar al cliente, no se enviará la relación y por tanto se parará la recursividad. Ejemplo:

```

public class Categoria {
    @Id
    @GeneratedValue (strategy=GenerationType.IDENTITY)
    private Long id;

    private String nombre;

    @OneToMany(fetch = FetchType.EAGER, mappedBy = "categoria", cascade = CascadeType.REMOVE)
    @JsonIgnore
    private List<Empleado> empleados = new ArrayList<>();

}

```

Gestión de errores

En todos los ejemplos anteriores hemos hecho una gestión muy sencilla de los errores y sin tratamiento basado en excepciones, simplemente construímos una respuesta vacía en la que asignábamos un código de respuesta (por ejemplo 404) para indicar el error, pero no dábamos al cliente ningún mensaje adicional sobre el error producido.

```
@GetMapping("/empleado/{id}")
public ResponseEntity<?> getOneElement(@PathVariable Long id) {
    Empleado empleado = empleadoService.obtenerPorId(id);
    if (empleado == null)
        return ResponseEntity.notFound().build(); // cod 404
    else
        return ResponseEntity.ok(empleado); // cod 200
}
```

En este apartado aprenderemos cómo gestionar los errores mediante excepciones y enviando al usuario una respuesta que además del código de error, informe con un texto del error producido. Desde la versión 3.2 de Spring, disponemos de la anotación **@RestControllerAdvice** para gestionar de forma centralizada en una sola clase todas las excepciones de toda una aplicación REST.

Por otra parte, desde Spring 5, también disponemos de una clase llamada **ResponseStatusException**, que ofrece una solución básica pero rápida para la gestión de errores, pero de forma individualizada, con una llamada para cada excepción. Empezamos por esta última que es más sencilla.

ResponseStatusException

ResponseStatusException es una clase hija de *RuntimeException* y tiene 3 constructores mediante los que podemos asignarle el estado a devolver (HttpStatus), opcionalmente el mensaje de error (String) y también opcionalmente la causa, por ejemplo, otra excepción (Throwable). Al invocarla, **enviamos al cliente la excepción producida, con su código de estado y mensaje**.

Un ejemplo sencillo de utilización sería este: supongamos un método del controlador que recibe un id de empleado y obtiene el empleado correspondiente del repositorio a través de un servicio. Si el servicio devuelve *null* cuando no encuentra el empleado buscado, podríamos lanzar una excepción con *ResponseStatusException*:

```
@GetMapping("/empleado/{id}")
public Empleado findById (@PathVariable Long id) {
    Empleado empleado = empleadoService.obtenerPorId(id);
    if (empleado == null)
        throw new ResponseStatusException(HttpStatus.NOT_FOUND, "Empleado no encontrado");
    return empleado;
}
```

De todas formas, esta no será la forma más correcta de trabajar, ya que el método de servicio en caso de error no debería de devolver *null* sino que debería de lanzar excepciones: **o bien lanzar *RuntimeException* o bien sus propias excepciones**, definidas por nosotros mismos, por ejemplo: *EmpleadoNotFoundException* si no encontramos un empleado, *EmptyEmpleadosException* si la base de datos de empleados está vacía, o cualquier otra situación de error que queramos contemplar.

Luego, desde el controlador **capturaremos esas excepciones y las llevaremos al cliente mediante *ResponseStatusException***. Este sistema nos da flexibilidad a la hora de configurar lo que llega al cliente, ya que es *ResponseStatusException* quién marca el código de estado y el mensaje, pudiendo ser distintos en diferentes controladores, aunque la excepción del servicio sea la misma.

El proceso a seguir sería el siguiente:

1.- Definir las excepciones, hijas de *RuntimeException* y con un constructor que puede recibir parámetros o no. El cuerpo del constructor solo invocará al constructor de *RuntimeException* con el mensaje asociado a esta excepción.

```
public class EmpleadoNotFoundException extends RuntimeException {  
    public EmpleadoNotFoundException(Long id) {  
        super("No se puede encontrar empleado con ID: " + id);  
    }  
}  
public class EmptyEmpleadosException extends RuntimeException {  
    public EmptyEmpleadosException() {  
        super("No hay empleados en el sistema");  
    }  
}
```

Cada excepción estará en su propio archivo .java y podemos meterlas todas en un paquete/carpeta llamado "exceptions" o algo similar.

2.- En el servicio, al llamar a los métodos del repositorio podemos invocar a las excepciones en caso de error:

```
public Empleado obtenerPorId(long id) {  
    Empleado empleado = repositorio.findById(id)  
        .orElseThrow(() -> new EmpleadoNotFoundException(id));  
    return empleado;  
}
```

La otra opción sería lanzar directamente *RuntimeException*, ahorrándonos la creación de la excepción:

```
public Empleado obtenerPorId(long id) {  
    Empleado empleado = repositorio.findById(id)  
        .orElseThrow(() -> new RuntimeException ("No se encuentra empleado con ID:" + id));  
    return empleado;  
}
```

Recordemos que el método *findById* de *JPARepository* devuelve un *Optional*, nunca devuelve nulo. *Optional* tiene un método *orElseThrow* que se ejecuta si el elemento dentro del *Optional* es nulo. Así pues, la sentencia anterior, si no encuentra el empleado se lanza la excepción, y si lo encuentra lo devuelve. Se puede escribir abreviado juntando el return y la llamada al método del repositorio:

```
public Empleado obtenerPorId (long id) {  
    return repositorio.findById(id).orElseThrow(() -> new EmpleadoNotFoundException(id));  
}
```

El formato "lambda" ()->... es debido al parámetro que necesita el método *orElseThrow*, no es debido a la llamada a la Excepción.

Podemos incluir la otra excepción definida, en otro método del servicio:

```
public List<Empleado> obtenerTodos() {  
    List<Empleado> lista = repositorio.findAll();  
    if (lista.isEmpty()) throw new EmptyEmpleadosException();  
    return lista; }
```

3.- En el controlador, capturamos las excepciones del servicio e invocamos a *ResponseStatusException* con el mensaje de la excepción definido en la primera excepción o bien con el mensaje que deseemos.

```
@GetMapping("/empleado/{id}")  
public Empleado getOneElement(@PathVariable Long id) {  
    try {  
        return empleadoService.obtenerPorId(id);  
    } catch (EmpleadoNotFoundException ex) {  
        throw new ResponseStatusException(HttpStatus.NOT_FOUND, ex.getMessage());  
    }  
}
```

Haríamos lo mismo con la otra excepción, la relativa a que no hay empleados en la base de datos. En el servicio:

```
@GetMapping("/empleado")
public List<Empleado> getList() {
    List<Empleado> listaEmpleados;
    try {
        listaEmpleados = empleadoService.obtenerTodos();
    } catch (EmptyEmpleadosException ex) {
        throw new ResponseStatusException(HttpStatus.NOT_FOUND, ex.getMessage());
    }
    return listaEmpleados;
}
```

Vemos la potencia de *ResponseStatusException* ya que en otro método del controlador podríamos enviar ante la misma excepción de servicio, otro status al cliente, por ejemplo, *BAD_REQUEST*.

4.- En los métodos *@PostMapping* y *@PutMapping*, si hay errores en los parámetros de entrada, que no se ajustan a los tipos de datos esperados, se producirá una excepción con código de estado 400 (*BAD_REQUEST*) sin necesidad de que nosotros programemos nada. Si además el objeto enviado tiene restricciones de validación (*@Min*, *@Email*, *@NotEmpty*, etc.) deberemos haber precedido el *@RequestBody* por ***@Valid*** para que se produzca la respuesta *BAD_REQUEST*. Sin *@Valid*, se produciría una respuesta más imprecisa: código de estado 500 : Internal Server Error.

5.- Para que todo esto funcione bien, debemos configurar tres aspectos más:

- Desde las últimas versiones, por seguridad, no se envía al usuario el mensaje de error (segundo parámetro del *ResponseStatusException*). Esto es así para reducir el riesgo de que le llegue al cliente información no deseada. Para cambiar este comportamiento, en *application.properties* añadimos: ***server.error.include-message=always***
- Con la dependencia DevTools añadida, la respuesta enviada al cliente incluye un campo llamado "trace" que incorpora información del servidor que es mejor ocultar a los clientes, por seguridad. Añadimos a *application.properties* la siguiente línea (o bien eliminamos la dependencia DevTools al pasar a producción nuestra aplicación): ***server.error.include-stacktrace=never***
- Internamente, al producirse una excepción de este tipo, accede a una ruta */error* aunque realmente no existe esa URL. Entonces, si tenemos configurada la seguridad como vimos en el capítulo anterior, debemos permitir el acceso a esta ruta en el bean *SecurityFilterChain*:
.requestMatchers("/error").permitAll()

Podemos probar con Postman, a solicitar un empleado que no existe:

GET localhost:9000/empleado/2

Params Auth Headers (6) Body Pre-req. Tests Settings Cookies Beautify

raw JSON

1

Body

404 Not Found 39 ms 323 B Save Response

Pretty Raw Preview Visualize JSON

```
1
2 "timestamp": "2022-09-28T20:02:11.347+00:00",
3 "status": 404,
4 "error": "Not Found",
5 "message": "No se puede encontrar empleado con ID: 2",
6 "path": "/empleado/2"
```

O la lista de empleados cuando no hay ninguno en la base de datos:

The screenshot shows a REST client interface. At the top, a 'GET' method is selected, and the URL 'localhost:9000/empleado/' is entered. Below the URL, there are tabs for 'Params', 'Auth', 'Headers (6)', 'Body', 'Pre-req.', 'Tests', 'Settings', and 'Cookies'. Under 'Params', there is a table with one row: 'Key' (Value) and 'Value' (Description). The 'Body' tab is selected, showing a JSON response with the following content:

```

1
2   "timestamp": "2022-09-28T19:24:41.725+00:00",
3   "status": 404,
4   "error": "Not Found",
5   "message": "No hay empleados en el sistema",
6   "path": "/empleado/"
7
  
```

At the top of the response body, there is a status bar with '404 Not Found 18 ms 312 B' and a 'Save Response' button. Below the response body, there are tabs for 'Pretty', 'Raw', 'Preview', 'Visualize', and 'JSON' (which is currently selected), along with a search bar.

Nos puede llamar la atención en estos ejemplos que el controlador no devuelve un `ResponseEntity`, sino un `Empleado` o un `List<Empleado>`. La respuesta es que en caso de que todo vaya bien (sin que se hayan producido excepciones) al devolver un objeto Java, se construye de forma automática la respuesta con el archivo JSON para el cuerpo y con código de respuesta 200. Es en caso contrario, cuando ya `ResponseStatusException` genera la respuesta con el código y mensaje adecuado.

RestControllerAdvice

La forma de tratar las excepciones que acabamos de ver no es centralizada, es decir, tenemos que incorporar la gestión de la excepción en cada método en el que pueda ocurrir llevándonos a duplicar código en algunos casos y a una organización menos controlada en caso de que tengamos muchos puntos en nuestro código en el que controlar las excepciones.

La clase anotada con `@ControllerAdvice` es una clase que contendrá todos los métodos para tratar todas las excepciones que se produzcan en cualquier método de servicio y lleguen a los controladores de nuestra aplicación. `@RestControllerAdvice` es una especialización de `@ControllerAdvice` que incluye un `@ResponseBody`, y será la que emplearemos nosotros.

Cada método de esta clase tendrá que incluir una anotación llamada `@ExceptionHandler` y el nombre de la excepción que gestiona con `.class`. Siguiendo con nuestro ejemplo:

```

@ExceptionHandler(EmpleadoNotFoundException.class)
public ResponseEntity<?> handleEmpleadoNotFound(Long id){
    return ResponseEntity.status(HttpStatus.NOT_FOUND).body(body)
}

@ExceptionHandler(EmptyEmpleadosException.class)
public ResponseEntity<?> handleEmptyEmpleados(Long id){
    return ResponseEntity.status(HttpStatus.NOT_FOUND).body(body);
}
  
```

En vez de crear la clase desde cero podemos declararla como hija de `ResponseEntityExceptionHandler` y aprovecharemos toda su funcionalidad. Entre todos los métodos que tiene esta superclase, es muy útil sobreescribir el método `handleExceptionInternal` que nos permite customizar el cuerpo por defecto de todas las excepciones que se puedan producir.

Así podemos tratar por una parte nuestras excepciones específicas y por otra el resto de excepciones que se puedan producir. Podríamos sobreescribir otros como `handleMissingPathVariable` o `handleTypeMismatch`.

Podemos crearla en cualquier punto de la aplicación, quizás el paquete `exceptions` sea un buen lugar. Entonces, los pasos a seguir para incluir la gestión de errores centralizada con `@RestControllerAdvice` serían los siguientes:

1.- Definir las excepciones que queremos que sean consideradas en nuestra aplicación, al igual que hicimos el caso de `ResponseStatusException`, hijas de `RuntimeException`.

```
public class EmptyEmpleadosException extends RuntimeException {
    public EmptyEmpleadosException() {
        super("No hay empleados en el sistema");
    }
}
```

2.- Crear la clase anotada con `@RestControllerAdvice`, hija de `ResponseEntityExceptionHandler` como se muestra a continuación. Un buen lugar sería el mismo paquete/carpeta donde hayamos situado las excepciones del punto anterior.

```
@RestControllerAdvice
public class GlobalControllerAdvice extends ResponseEntityExceptionHandler {
    @ExceptionHandler(EmpleadoNotFoundException.class)
    public ResponseEntity<?> handleEmpleadoNotFound(
        EmpleadoNotFoundException ex, WebRequest request) {
        ExpcionBody body =
            new ExpcionBody(LocalDateTime.now(),
                HttpStatus.NOT_FOUND, ex.getMessage(),
                ((ServletWebRequest) request).getRequest().getRequestURI());
        return new ResponseEntity<Object>(body, HttpStatus.NOT_FOUND);
    }

    @ExceptionHandler(EmptyEmpleadosException.class)
    public ResponseEntity<?> handleEmptyEmpleados(
        EmptyEmpleadosException ex, WebRequest request) {
        ExpcionBody body =
            new ExpcionBody(LocalDateTime.now(),
                HttpStatus.NOT_FOUND, ex.getMessage(),
                ((ServletWebRequest) request).getRequest().getRequestURI());
        return new ResponseEntity<Object>(body, HttpStatus.NOT_FOUND);
    }

    @Override
    protected ResponseEntity<Object> handleExceptionInternal(
        Exception ex, @Nullable Object body, HttpHeaders headers, //(*)
        HttpStatusCode status, WebRequest request) {
        ExpcionBody myBody =
            new ExpcionBody(LocalDateTime.now(),
                status, ex.getMessage(),
                ((ServletWebRequest) request).getRequest().getRequestURI());
        return ResponseEntity.status(status).headers(headers).body(myBody);
    }
}
```

El import de `Nullable` no es de Lombok, es de: `org.springframework.lang`

Y definimos en este mismo archivo la clase `ExpcionBody`(sin public), con los atributos que queremos que se devuelvan al cliente en la respuesta JSON :

```
@AllArgsConstructor
@Getter
class ExpcionBody {
    private LocalDateTime timestamp;
    private HttpStatusCode status;
    private String message;
    private String path;
}
```

3.- Los métodos de los controladores ahora son mucho más sencillos que en el caso de `ResponseStatusException`, e incluso también más sencillos que en los que hacíamos al principio de este capítulo, ya que no tenemos que tener en cuenta para nada si se produce un error o no.

Por ejemplo, un método `@GetMapping` podría ser así de sencillo, ya que la respuesta por defecto es con código 200, OK y el resto de situaciones con sus códigos de estado se traspasan al gestor de excepciones.

```
@GetMapping("/empleado")
public List<Empleado> getList() {
    List<Empleado> listaEmpleados = empleadoService.obtenerTodos();
    return listaEmpleados;
}
```

En el caso de `@PostMapping` podemos validar los datos enviados con las anotaciones de validación incluidas en la entidad (`@Min`, `@Max`, `@NotEmpty`, `@Email`, etc.) de forma que devuelva un error 400 `BAD_REQUEST`. Esto lo lograremos simplemente añadiendo la anotación `@Valid` a los datos enviados, esto es, al `@RequestBody`.

Por otra parte, no queremos un código 200 de respuesta, queremos 201, por lo que sería algo así:

```
@PostMapping("/empleado")
public ResponseEntity<?> newElement(@Valid @RequestBody Empleado nuevoEmpleado) {
    Empleado empleado = empleadoService.añadir(nuevoEmpleado);
    return ResponseEntity.status(HttpStatus.CREATED).body(empleado); // cod 201
}
```

El caso del `@PutMapping` para modificar un elemento es similar a `@PostMapping`, pero antes de modificar el objeto, podemos añadir una llamada al `findById` de dicho objeto, para verificar que el elemento que vamos a modificar existe. Si no existe se producirá una excepción `404 NOT_FOUND`:

```
@PutMapping("/empleado/{id}")
public Empleado editElement(@Valid @RequestBody Empleado editEmpleado,
    @PathVariable Long id) {
    empleadoService.obtenerPorId(id); // esto es para ver si se produce excepción
    return empleadoService.editar(editEmpleado);
}
```

A tener en cuenta:

- Podríamos tener más de una clase `@ControllerAdvice` o de su especialización `@RestControllerAdvice` en nuestra aplicación, aunque no es frecuente, lo habitual es que una sola clase agrupe la gestión de todas las excepciones.
- Por defecto, las clases anotadas con `@ControllerAdvice` / `@RestControllerAdvice` van a controlar las excepciones que lleguen a cualquier método de cualquier controlador de la aplicación, pero se podría restringir para que tratase las de algún solo controlador, alguna sola clase, etc. Ejemplo:
`@RestControllerAdvice("my.chosen.package")`

Elementos avanzados

Envío de ficheros (Opcional)

La recepción de archivos en nuestro servidor procedentes del cliente ya fue tratada en temas anteriores, pero bajo el esquema Spring MVC. Para las aplicaciones REST es válido todo lo que comentamos en aquel capítulo en cuanto al servicio de recepción y almacenamiento de archivos (`FileStorageService`), pero obviamente, el controlador será distinto y también la forma en la que nuestros clientes enviarán los archivos.

Para subir archivos en nuestros métodos del controlador (@PostMapping y PutMapping) tendremos que indicarle que lo que recibimos son datos *multiparte*, es decir, no son solo datos en formato JSON inyectables con @RequestBody si no que tendremos distintos tipos de datos. Para ello emplearemos la anotación **@RequestPart** (podríamos usar @RequestParm, pero nos implicaría algún proceso de conversión de formatos adicional). Ejemplo:

```
@PostMapping(value="/nuevoEmpleado", consumes=MediaType.MULTIPART_FORM_DATA_VALUE)
public ResponseEntity<?> newElement(@RequestPart("data") Empleado nuevoEmpleado,
                                     @RequestPart("file") MultipartFile file) {
```

En este ejemplo, el parámetro *value* es la ruta a la que responderá el controlador (lo que hasta ahora poníamos habitualmente sin *value*) y el parámetro *consumes*, opcional, indica el tipo de dato que recibiremos.

A continuación, mediante la anotación **@RequestPart** le indicamos que recibiremos:

- **@RequestPart("data") Empleado nuevoEmpleado**: datos textuales en un elemento llamado "data" y que mapearemos en una instancia de una clase de nuestro dominio (Empleado en este caso) y por defecto en formato JSON.
- **@RequestPart("file") MultipartFile file**: Archivo que puede ser una imagen, un PDF, etc. Los nombres "data" y "file" podemos decidirlo nosotros, pero deberá conocerlos la aplicación cliente que envíe los datos.

CRUD de Empleado con imagen

Vamos a ver cómo sería el proceso de modificar nuestro proyecto de CRUD de Empleado con API REST para añadir un archivo al empleado, por ejemplo, una imagen. Los pasos serían:

1.- Añadir un nuevo campo a la entidad Empleado de tipo String, que almacén el nombre de la imagen. También habría que modificar los DTO relacionados, si existiesen.

2.- Desarrollar un servicio FileStorageService igual al del capítulo 5 y crear en la raíz del proyecto la carpeta *uploadDir*. El servicio sería así:

```
@Service
public class FileStorageService {
    private final Path rootLocation = Paths.get("uploadDir");

    public String store(MultipartFile file) throws Exception {
        String filename = StringUtils.cleanPath(file.getOriginalFilename());
        String extension = StringUtils.getFilenameExtension(filename);
        String storedFilename = System.currentTimeMillis() + "." + extension;

        if (file.isEmpty()) throw new Exception("archivo enviado vacío");
        if (filename.contains("..")) throw new Exception("nombre de archivo incorrecto");

        try (InputStream inputStream = file.getInputStream()) {
            Files.copy(inputStream, this.rootLocation.resolve(storedFilename),
                       StandardCopyOption.REPLACE_EXISTING);
            return storedFilename;
        } catch (IOException ioe) {throw new Exception("Error al almacenar el archivo");}
    }

    public Resource loadAsResource(String filename) {
        try {
            Path file = rootLocation.resolve(filename);
            Resource resource = new UrlResource(file.toUri());
            if (resource.exists() || resource.isReadable()) { return resource;
            } else { throw new Exception(); }
        } catch (Exception e) { System.err.println("Error IO"); }
        return null;
    }
}
```

```

public boolean delete(String filename) {
    try {
        Path file = rootLocation.resolve(filename);
        if (Files.exists(file)) {
            Files.delete(file);
            return true;
        } else return false;
    } catch (IOException ioe) { return false; }
}
}

```

3.- Modificar el RestController para que, en los casos de POST y PUT, sea capaz de recibir dos parámetros distintos desde el cliente, por una parte, los datos del empleado en formato JSON (como en los ejemplos anteriores) y por otra parte el fichero, tal y como acabamos de explicar en el apartado anterior.

```

@RestController
public class EmpleadoController {
    @Autowired
    private EmpleadoService empleadoService;

    @Autowired
    private FileStorageService fileStorageService;

    @GetMapping("/empleado") {
        . . .
    }
    @GetMapping("/empleado/{id}") {
        . . .
    }

    @PostMapping(value="/empleado", consumes=MediaType.MULTIPART_FORM_DATA_VALUE)
    public ResponseEntity<?> newElement(@RequestPart("data") Empleado nuevoEmpleado,
                                         @RequestPart("file") MultipartFile file) {
        if (!file.isEmpty()) {
            try { nuevoEmpleado.setImagen(fileStorageService.store(file));
            } catch (Exception e) { nuevoEmpleado.setImagen(null); }
        } else { nuevoEmpleado.setImagen(null); }
        Empleado empleado = empleadoService.agregar(nuevoEmpleado);
        return ResponseEntity.status(HttpStatus.CREATED).body(empleado);
    }

    @PutMapping(value = "/empleado/{id}", consumes = MediaType.MULTIPART_FORM_DATA_VALUE)
    public ResponseEntity<?> editElement(@RequestPart("data") Empleado editEmpleado,
                                         @RequestPart("file") MultipartFile file,
                                         @PathVariable Long id) {
        Empleado empleado = empleadoService.obtenerPorId(id);
        if (empleado == null) return ResponseEntity.notFound().build(); // cod 404
        else {
            if (!file.isEmpty()) {
                try {
                    editEmpleado.setImagen(fileStorageService.store(file));
                } catch (Exception e) { editEmpleado.setImagen(null); }
            } else { editEmpleado.setImagen(null); }
            empleado = empleadoService.editar(editEmpleado);
            return ResponseEntity.ok(empleado); // cod 200
        }
    }
}

```

4.- En el RestController también añadimos un método para enviar los ficheros cuando sean solicitados.

```

@GetMapping(value = "/files/{filename:.+}")
public ResponseEntity<Resource> serveFile(@PathVariable String filename,
                                         HttpServletRequest request) {
    Resource file = fileStorageService.loadAsResource(filename);
    String contentType = null;
    try { contentType = request.getServletContext().getMimeType
           (file.getFile().getAbsolutePath());
    } catch (IOException ex) {System.err.println("Could not determine file type.");}
    if (contentType == null) contentType = "application/octet-stream";
    return ResponseEntity.ok()
        .contentType(MediaType.parseMediaType(contentType))
        .body(file);
}

```

Probando desde Postman

En este caso el cuerpo de la petición no será de tipo JSON, será de tipo **form-data** e incluiremos dos pares de clave valor según se ve en la imagen siguiente, por una parte, la clave "data" con un archivo JSON con los datos textuales del empleado, y por otra parte la clave "file" con el archivo de imagen. Los nombres de las claves se corresponden con los empleados en los parámetros @RequestPart del controlador.

Body

KEY	VALUE	DESCRIPTION	...	Bulk Edit
<input checked="" type="checkbox"/> data	nuevoEmpleado.json			
<input checked="" type="checkbox"/> file	avatar1.png			
Key	Value	Description		

```

1  {
2      "id": 8,
3      "nombre": "nuevoConImagen",
4      "email": "imagen@gmail.com",
5      "salario": 800.0,
6      "enActivo": true,
7      "genero": "MASCULINO",
8      "Imagen": "1662893730712.png"
9  }

```

HATEOAS

Este acrónimo se corresponde con "Hypermedia As The Engine Of Application State" y viene a decir que el cliente debe poder moverse por la aplicación web únicamente siguiendo a través de hipervínculos de las respuestas recibidas, sin necesidad de crear peticiones nuevas independientes.

Un ejemplo del estándar HATEOAS podría ser que, si una petición me devuelve un archivo JSON con los nombres de los empleados de una empresa, esta respuesta debería incluir no solo el nombre del empleado, sino que también un enlace para cada uno de ellos que nos devolviese el detalle de cada empleado. Si además para cada empleado, la respuesta ofrece el departamento al que pertenece, también debería incluir un enlace a los datos de ese departamento. De esta forma se podría navegar por toda la aplicación saltando de enlace en enlace sin tener que elaborar nuevas peticiones independientes.

También se puede configurar para facilitar al cliente el saber qué operaciones puede hacer y cuales no puede hacer, introduciendo en la respuesta enlaces para eliminar el recurso, actualizarlo, etc.

En la parte negativa está que HATEOAS añade complejidad a la API, que afecta tanto al desarrollador de la API como al consumidor de la misma. Hay que realizar un trabajo adicional para añadir los enlaces apropiados en cada respuesta según el estado de la entidad. Esto provoca que la API sea más compleja de construir que una API que no implementa HATEOAS. Los clientes de la API también tienen complejidad añadida para entender la semántica de cada enlace además de tener y procesar cada respuesta para obtener los enlaces.

```
< → C ⓘ http://localhost:9000/empleados

{
  "_embedded" : {
    "empleados" : [ {
      "nombre" : "pepe",
      "email" : "pepe@gmail.com",
      "salario" : 800.0,
      "enActivo" : true,
      "genero" : "MASCULINO",
      "links" : {
        "self" : {
          "href" : "http://localhost:9000/empleados/1"
        },
        "empleado" : {
          "href" : "http://localhost:9000/empleados/1"
        },
        "departamento" : {
          "href" : "http://localhost:9000/empleados/1/departamento"
        }
      }
    },
    {
      "nombre" : "ana",
      "email" : "ana@gmail.com",
      "salario" : 900.0,
      "enActivo" : true,
      "genero" : "FEMENINO",
      "links" : {
        "self" : {
          "href" : "http://localhost:9000/empleados/2"
        },
        "empleado" : {
          "href" : "http://localhost:9000/empleados/2"
        },
        "departamento" : {
          "href" : "http://localhost:9000/empleados/2/departamento"
        }
      }
    },
    {
      "nombre" : "luis"
    }
  }
}
```

Vemos en la imagen anterior que cada elemento tiene un link "self" que enlaza al propio elemento y otros links como "departamento" que enlazan con relaciones del recurso con otras clases.

Spring nos ofrece tres clases para incorporar estos enlaces a las representaciones de nuestros recursos:

- **RepresentationModel**:clase base para que nuestras clases de dominio (o mejor aún, nuestros DTOs) incorporen los enlaces.
 - **Link**: Clase cuyas instancias contienen tanto el hipervínculo, así como la relación del enlace con el recurso, por ejemplo “self”.
 - **WebMvcLinkBuilder**: Permite construir instancias de Link que incluyan los mappings de los controladores de nuestra aplicación.

HAL

A veces se confunde el término HATEOAS con HAL (*Hypertext Application Language*) y en realidad son complementarios. Para implementar HATEOAS es necesario un lenguaje que represente los recursos que contendrán hipervínculos para navegar por la aplicación. HAL es uno de esos lenguajes, es decir, un formato específico para la representación de recursos (con enlaces) pero podría emplearse cualquier otro lenguaje para implementar HATEOAS.

Los niveles de madurez REST

Son distintos niveles de utilización de la semántica del protocolo HTTP y, en cierto modo, una medida de calidad de los servicios web ofrecidos por nuestra aplicación.

- Nivel 0, *transporte HTTP*. Usa HTTP como medio de transporte, pero sin usar la semántica de la web. Cada petición tiene su propia dirección de *endpoint*.
- Nivel 1, *recursos*: cada recurso tiene su propia dirección web, endpoint o URL.
- Nivel 2, *verbos*: Se usan los verbos HTTP (GET, PUT, POST, DELETE) para las operaciones CRUD.
- Nivel 3, *controles hipermédia*: permite la navegación (HATEOAS)

Añadir HATEOAS al proyecto

Para que las respuestas de nuestra aplicación API REST incluyan la navegación por enlaces de la que estamos hablando debemos seguir los siguientes pasos:

1.- Incorporar al pom.xml la dependencia starter HATEOAS:

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

2.- Las clases de modelo de dominio (o DTOs) que pretendemos adaptar deberán ser hijas de la clase *RepresentationModel* para poder usar los métodos que nos ayudarán a crear los enlaces del recurso. Heredamos un método **add()** con el que añadiremos los enlaces a la representación del recurso sin tener que añadir nuevos atributos a la clase.

```
public class Empleado extends RepresentationModel<Empleado> { . . . }
```

3.- Creamos los links que deseamos añadir a la representación con la clase *Link*. El siguiente ejemplo crearía un enlace estático (no es lo que buscamos):

```
Link link = Link.of("http://localhost:9000/empleado/1");
Empleado empleado.add(link);
```

Para crear enlaces dinámicos (no en hard-code) podemos usar *WebMvcLinkBuilder* que simplifica esta tarea. El siguiente ejemplo crea el enlace "self" de un empleado:

```
Link link = WebMvcLinkBuilder.linkTo(EmpleadoController.class)
    .slash(empleado.getId()).withSelfRel();
Empleado empleado.add(link);
```

Así, ante la URL: <http://localhost:9000/empleado/2>, obtendríamos algo como:

```
{
    "id": 2,
    "nombre": "ana",
    "email": "ana@gmail.com",
    "salario": 900.0,
    "_links": {
        "self": {
            "href": "http://localhost:9000/empleado/2"
        }
    }
}
```

Podríamos añadir otro enlace para navegar a todas las cuentas (<http://localhost:9000/empleados>)

```
Link link2 = WebMvcLinkBuilder.linkTo(CuentaController.class)
    .slash("empleados").withRel("all");
Empleado empleado.add(link2);
```

Obteniendo:

```
{
  "id": 2,
  "nombre": "ana",
  "email": "ana@gmail.com",
  "salario": 900.0,
  "_links": {
    "self": {
      "href": "http://localhost:9000/empleado/2"
    },
    "all": {
      "href": "http://localhost:9000/empleados"
    }
  }
}
```

Estas operaciones se harán típicamente en el controlador que va devolver el objeto o DTO.

4.- Podríamos crear enlaces para las relaciones del recurso con otras clases, por ejemplo, para un empleado con su Departamento o con los Proyectos con los que se relaciona. Para estos últimos (1 a muchos) deberíamos usar la clase *CollectionModel*.

Spring Data Rest

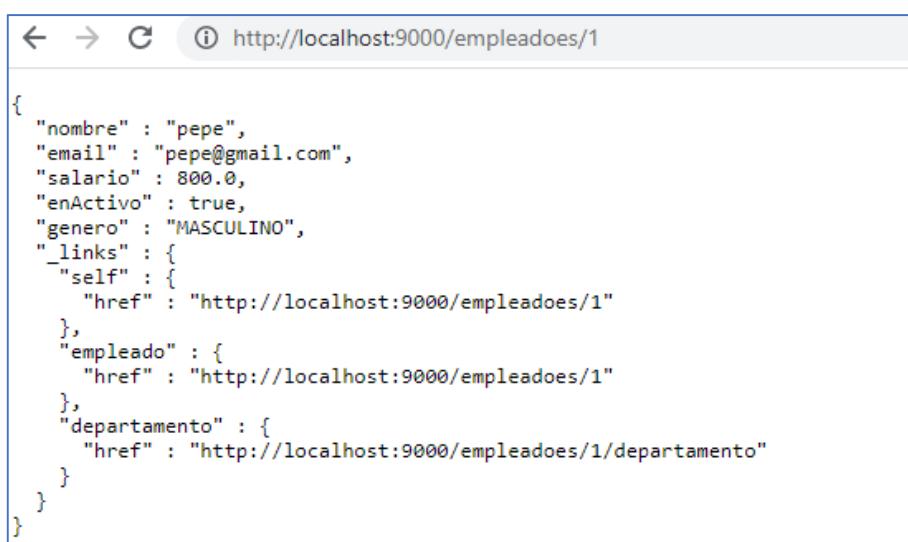
Aunque hemos visto que no es complicado crear un servicio API Rest sobre un determinado repositorio, sí que es cierto que hay que escribir cierta cantidad de código para crear todos los métodos CRUD del servicio y asociarlos al repositorio correspondiente.

Spring Data Rest es un módulo de Spring que nos permite crear un servicio de forma transparente para nosotros con las operaciones típicas CRUD de un repositorio JPA dado, con soporte hipermedia (HATEOAS + HAL) de forma automática y sin escribir una línea de código, tan solo incorporando una anotación.

En este caso no es necesario incluir ningún servicio ni controlador en la aplicación; esto aporta sencillez, pero no permite customizar el acceso como, por ejemplo, añadir lógica de negocio en el servidor.

Podríamos configurar su comportamiento en detalle, pero vamos a verlo en funcionamiento con sus parámetros por defecto. Para que funcione, solo tenemos que añadir a nuestro proyecto la dependencia **starter-data-rest** (y opcionalmente HAL Explorer).

Solo con hacer esto, nuestras respuestas ya cumplen el estándar HATEOAS. Si lo probamos sobre uno de los ejemplos anteriores, el CRUD de Empleado con asociación @ManyToOne con Departamento, se generarán nuevas rutas, formadas por los "plurales a la inglesa" de las entidades (*empleados*, *departamentoes*) con el formato HATEOAS.



En la imagen anterior podemos ver como disponemos de un enlace a el mismo y también un enlace a su departamento.

Las anotaciones `@RepositoryRestResource` para Repositorios y `@RestResource` para Entidades son opcionales y permiten modificar este comportamiento por defecto, por ejemplo, para excluir ciertos repositorios y que no se comporten de esta forma, y se sigan mostrando como antes.

Un parámetro que también podemos modificar es la ruta en plural que va a generar, para que lo haga correctamente, es decir, empleados en vez de `empleados` y departamentos en vez de `departamentos`.

```
@RepositoryRestResource(path = "empleados", collectionResourceRel = "empleados")
public interface EmpleadoRepository extends JpaRepository<Empleado, Long> { }
```

Por último, comentar que si añadimos la dependencia HAL Explorer también se genera un explorador de la API del proyecto para realizar operaciones sobre la misma. Se accede a ella con `http://localhost:puerto`

The screenshot shows the HAL Explorer interface at `http://localhost:9000/explorer/index.html#uri=/`. The left panel displays a table of links with columns: Relation, Name, Title, HTTP Request, and Doc. The relations listed are `empleados`, `departamentos`, and `profile`. The right panel shows the Response Status (200 OK), Response Headers (connection: keep-alive, content-type: application/hal+json, date: Tue, 13 Sep 2022 15:42:47 GMT, keep-alive: timeout=60, transfer-encoding: chunked, vary: Origin, Access-Control-Request-Method, Access-Control-Request-Headers), and the Response Body, which is a JSON object representing the links for the three relations.

El proceso a seguir sería tan sencillo como:

1.- Añadir la dependencia starter Data Rest al pom.xml (además de JPA y la del gestor de base de datos como pueda ser H2):

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

Y opcionalmente:

```
<dependency>
  <groupId>org.springframework.data</groupId>
  <artifactId>spring-data-rest-hal-explorer</artifactId>
</dependency>
```

2.- Si queremos corregir los plurales (`empleados` en vez de `empleados`), podemos añadir la anotación `@RepositoryRestResource` al repositorio:

```
@RepositoryRestResource(collectionResourceRel = "empleados", path = "empleados")
public interface EmpleadoRepository extends JpaRepository<Empleado, Long> { }
```

y tendremos un servicio totalmente funcional con un entry point llamado `/empleados`.

OpenAPI y Swagger

Como hemos comentado, una aplicación o servicio API REST ofrece una serie de recursos para ser consumidos mediante HTTP por diferentes clientes, estas pueden ser aplicaciones cliente finales, o bien otras aplicaciones de servidor, en cualquiera de los casos, **serán otros programadores, no usuarios finales**, los que harán uso de nuestra API por lo que una buena documentación es fundamental: funcionalidad de la aplicación, qué verbos HTTP emplear, que parámetros se requieren, que valores se devuelven.

Open API es una especificación, esto es una serie de reglas para documentar todos estos aspectos de nuestra API y es el estándar de facto utilizado actualmente. Swagger es una herramienta que nos permite construir y consultar esa documentación y acceder a ella de una forma sencilla (a veces se confunden los términos Open API y Swagger). Open API trabaja a partir de un archivo, por defecto JSON con toda la documentación del API, y Swagger, mediante su herramienta SwaggerUI, permite un acceso en entorno gráfico de forma que no solo podremos consultar la documentación, sino que podremos probar su funcionalidad.

En <https://petstore.swagger.io/> podemos ver un ejemplo en funcionamiento:

The screenshot shows the Swagger Petstore UI. At the top, it displays the URL <https://petstore.swagger.io/v2/swagger.json> and an 'Explore' button. Below this, the title 'Swagger Petstore 1.0.6' is shown, along with a note that it's a sample server. It includes links for 'Terms of service', 'Contact the developer', 'Apache 2.0', and 'Find out more about Swagger'. A dropdown menu for 'Schemes' is set to 'HTTPS'. The main content area shows the 'pet' resource with the following operations:

- pet** Everything about your Pets
- Find out more**
- GET /pet/{petId}** Find pet by ID
- POST /pet/{petId}** Updates a pet in the store with form data
- DELETE /pet/{petId}** Deletes a pet (highlighted in red)
- POST /pet/{petId}/uploadImage** uploads an image
- POST /pet** Add a new pet to the store
- PUT /pet** Update an existing pet
- GET /pet/findByStatus** Finds Pets by status

La parte más tediosa de Swagger podría ser la de construir ese archivo "json" origen con toda la información de la API. Este sería un ejemplo de ese tipo de archivo:

```

1  {
2      "swagger": "2.0",
3      "info": {
4          "description": "This is a sample server Petstore server. You can find out more about Swagger at [http://swagger.io]. The server is running locally on port 8080 and the operations are all exposed on /api/v2 path prefix. Try them out by visiting [http://petstore.swagger.io/v2/swagger.json]. Swagger UI is served from this same endpoint. Try out the operation below! If you break it, you'll get helpful error information. You can build a custom API and reference it here. You will need to add it to the list of external documents below.", "version": "1.0.6", "title": "Swagger Petstore", "termsOfService": "http://swagger.io/terms/", "contact": { "email": "apiteam@swagger.io" }, "license": { "name": "Apache 2.0", "url": "http://www.apache.org/licenses/LICENSE-2.0.html" } },
5      "host": "petstore.swagger.io",
6      "basePath": "/v2",
7      "tags": [
8          {
9              "name": "pet",
10             "description": "Everything about your Pets",
11             "externalDocs": {
12                 "description": "Find out more",
13                 "url": "http://swagger.io"
14             }
15         }
16     ]
17 }

```

Como ya supondréis, no tendremos que construir “a mano” ese archivo, será Spring quién lo construirá por nosotros mediante la librería **SpringDoc**, tal y como veremos a continuación. Por una parte, SpringDoc, generará la estructura básica de la documentación de nuestra API y por otra, nos ofrecerá clases para afinar aún más esa documentación.

Incorporar Open API al proyecto

1.- Añadir la dependencia SpringDoc: **springdoc-openapi-starter-webmvc-ui** (esto es para la versión 3, para anteriores eran necesarias otras dependencias).

```
<dependency>
    <groupId>org.springdoc</groupId>
    <artifactId>springdoc-openapi-starter-webmvc-ui</artifactId>
    <version>2.2.0</version>
</dependency>
```

2.- Solo por añadir la dependencia anterior, en la ruta **/v3/api-docs** tendremos acceso a la documentación de nuestra api en formato JSON. Podemos cambiar esta ruta por defecto a la que deseemos, con el siguiente parámetro en el fichero *application.properties*. Ejemplos:

```
springdoc.api-docs.path=/api-docs
```

3.- La dependencia incluye Swagger, por lo que en la ruta **http://localhost/swagger-ui/index.html** podremos visualizarla e interactuar con ella.

También podremos cambiar en el *application.properties* las conocidas como *Swagger-ui properties* como la ruta de la documentación en formato Swagger, o si incluimos los *endpoints* de *Actuator* en la documentación Swagger.

```
springdoc.swagger-ui.disable-swagger-default-url=true
springdoc.swagger-ui.path=/mydoc
springdoc.swagger-ui.tryItOutEnabled=true
springdoc.show-actuator=true
```

Si accedemos a la documentación generada, comprobaremos que tendrá muchos valores por defecto, como el título, descripción, etc.

4.- Podemos crear una clase de configuración que incluya información genérica de nuestra API: título, descripción, versión, contacto, etc:

```
import io.swagger.v3.oas.models.OpenAPI;
import io.swagger.v3.oas.models.info.Contact;
import io.swagger.v3.oas.models.info.Info;
import io.swagger.v3.oas.models.info.License;
import io.swagger.v3.oas.models.servers.Server;

@Configuration
public class OpenApiConfig {
    @Bean
    public OpenAPI myOpenAPI() {
        Server prodServer = new Server();
        prodServer.setUrl("http://localhost:9000");
        prodServer.setDescription("Server URL in Production environment");

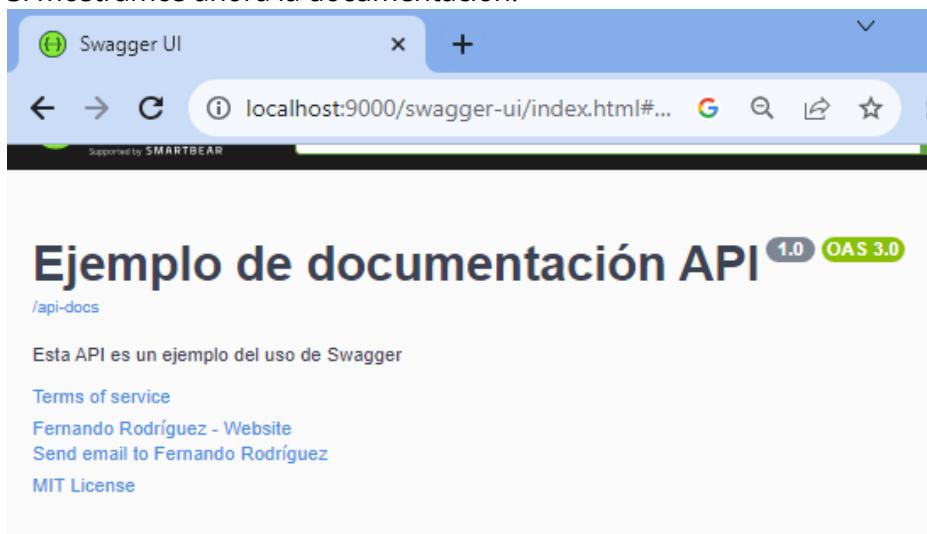
        Contact contact = new Contact();
        contact.setEmail("rdf@fernandowirtz.com");
        contact.setName("Fernando Rodríguez");
        contact.setUrl("https://www.fernandowirtz.com");

        License mitLicense =
            new License().name("MIT License").url("https://choosealicense.com/licenses/mit/");

        Info info = new Info()
            .title("Ejemplo de documentación API")
            .version("1.0")
            .contact(contact)
    }
}
```

```
        .description("Esta API es un ejemplo del uso de Swagger")
        .termsOfService("https://www.fernandowirtz.com/terms")
        .license(mitLicense);
    return new OpenAPI().info(info).servers(List.of(prodServer));
}
```

Si mostramos ahora la documentación:



5.- Además de esta documentación general de la API disponemos de anotaciones que permiten afinar

```
@Tag(name = "Nombre de la clase", description = "Descripción de la clase")
@RestController
@RequestMapping("/api/productos")
public class ProductoRestController {
```

A nivel de método de controlador disponemos de:

- `@Operacion`: describe qué hace el método del controlador.
 - `@ApiResponse/s`: describe las distintas respuestas que puede dar el método.
 - `@Parameter`: describe cada parámetro que recibe el método.

```
@Operation(summary = "resumen de la operación (del mapping)",
    description = "Descripción de la operación (del mapping).",
    tags = {"etiquetas calificadoras", "get" })
@ApiResponses({
    @ApiResponse(responseCode = "200",
        content = {@Content(schema = @Schema(implementation = Producto.class),
            mediaType = "application/json") }),
    @ApiResponse(responseCode = "404",
        content = { @Content(schema = @Schema()) }),
    @ApiResponse(responseCode = "500",
        content = { @Content(schema = @Schema()) })})

@GetMapping("/{id}")
public Producto getOne(
    @Parameter(name = "id",
        description = "identific. único del producto", example = "1", required = true)
    @PathVariable Long id) {
    return productoService.obtenerPorId(id);
}
```

GET /api/productos/{id} resumen de la operación (del mapping)

Descripción de la operación (del mapping).

Parameters

Name	Description
id * required	integer(\$int64) identificador único del producto (path)
	<input type="text" value="1"/>

Execute

Responses

Curl

```
curl -X 'GET' \
  'http://localhost:9000/api/productos/1' \
  -H 'accept: application/json'
```

Request URL

```
http://localhost:9000/api/productos/1
```

Server response

Code	Details
200	<p>Response body</p> <pre>{ "id": 1, "nombre": "Prodi", "enOferta": true, "tipoIva": "REDUCIDO", "precio": 1000, "categoria": { "id": 1, "nombre": "Cat 1" } }</pre> <p>Response headers</p> <pre>connection: keep-alive content-type: application/json date: Mon,13 Nov 2023 20:43:29 GMT keep-alive: timeout=60 transfer-encoding: chunked vary: Origin,Access-Control-Request-Method,Access-Control-Request-Headers</pre>

Responses

Code	Description
200	

En el siguiente enlace puedes ver todas las anotaciones Swagger disponibles:

<https://github.com/swagger-api/swagger-core/wiki/Swagger-2.X---OpenAPI-3.1>

Consumiendo APIs

Plataformas cliente

Como llevamos comentando a lo largo de este tema, las API Rest son aplicaciones que están a disposición de otras aplicaciones para que consuman los recursos a través del protocolo HTTP. Unos de los clientes más habituales serán aplicaciones web con JavaScript. Existen varias formas de llamar a una API Rest desde JavaScript. Estas serían las cuatro más habituales:

1.- **XMLHttpRequest**: es un estándar clásico y el elemento básico en el que se basa AJAX y otros frameworks y librerías:

```
var xhttp = new XMLHttpRequest();
xhttp.onreadystatechange = function() {
  if (this.readyState == 4 && this.status == 200) {
    console.log(JSON.parse(this.responseText));
  }
};

xhttp.open("GET", "http://localhost:9000/empleado", true);
xhttp.setRequestHeader("Content-type", "application/json");
xhttp.send(null);
```

2.- **Fetch**. Similar a XMLHttpRequest pero con una notación más sencilla y basada en promesas. Es la más empleada actualmente, apoyándonos en **async await** para gestionar esas promesas de una forma sencilla. Basándonos en el ejemplo desarrollado a lo largo del capítulo, que desarrolla un CRUD de Empleado, el siguiente código recuperaría los datos de los empleados y construiría el *body* de una tabla con los mismos.

```
async function obtenerEmpleados() {
  try {
    const response = await fetch('http://localhost:9000/empleado');

    const empleados = await response.json();

    document.getElementById('tablaEmpleadosBody').innerHTML = '';
    empleados.forEach(empleado => {
      agregarFilaATabla(empleado);
    });
  } catch (error) { console.error('Error al obtener empleados:', error); }
}

function agregarFilaATabla(empleado) {
  const tablaBody = document.getElementById('tablaEmpleadosBody');
  const fila = document.createElement('tr');
  fila.innerHTML = `
    <td>${empleado.id}</td>
    <td>${empleado.nombre}</td>
    <td>${empleado.salario}</td>

    <td>
      <button onclick="modificarEmpleado(${empleado.id})">Modificar</button>
      <button onclick="eliminarEmpleado(${empleado.id})">Borrar</button>
    </td>
  `;
  tablaBody.appendChild(fila);
}
```

El código para añadir un nuevo empleado podría ser así:

```

async function agregarEmpleado() {
  const nombre = document.getElementById('newNombre').value;
  const salario = document.getElementById('newSalario').value;
  const empleadoData = {
    id: 0,                                     //no lo informamos @GeneratedValue
    nombre: nombre,
    salario: salario
  };

  try {
    const response = await fetch(`http://localhost:9000/empleado`, {
      method: 'POST',
      headers: {
        'Content-Type': 'application/json'
      },
      body: JSON.stringify(empleadoData)
    });
    if (response.ok) {
      await obtenerEmpleados();
    } else { alert('Error al registrar empleado'); }
  } catch (error) {
    alert.error('Error en la solicitud:', error);
  }
}

```

3.- **JQuery**. Esta es una librería muy utilizada anteriormente, reemplazada hoy en día por las vistas previamente.

```

<!DOCTYPE html>
<html lang="en">
<head><meta charset="utf-8">
  <meta http-equiv="X-UA-Compatible" content="IE=edge">
  <meta name="viewport" content="width=device-width, initial-scale=1">
</head>
<body><h1>Listado de empleados</h1>
  <table>
    <thead><tr><th>Id</th><th>Nombre</th><th>Email</th></tr></thead>
    <tbody id="empleados-table-body"></tbody>
  </table>
<script src="https://ajax.googleapis.com/ajax/libs/jquery/1.12.4/jquery.min.js"></script>
<script>
$(document).ready(function(){
  $.ajax({
    url: 'http://localhost:9000/empleado/',
    type: 'GET',
    success: function(data) {
      var html = "";
      $.each(data, function(index, value){
        html += '<tr>';
        html += '<td>' + value.id + '</td>';
        html += '<td>' + value.nombre + '</td>';
        html += '<td>' + value.salario + '</td>';
        html += '</tr>';
      });
      $("#empleados-table-body").append(html);
    },
    error: function(error) {console.log("Error: " + error); }
  });
});
</script>
</body>
</html>

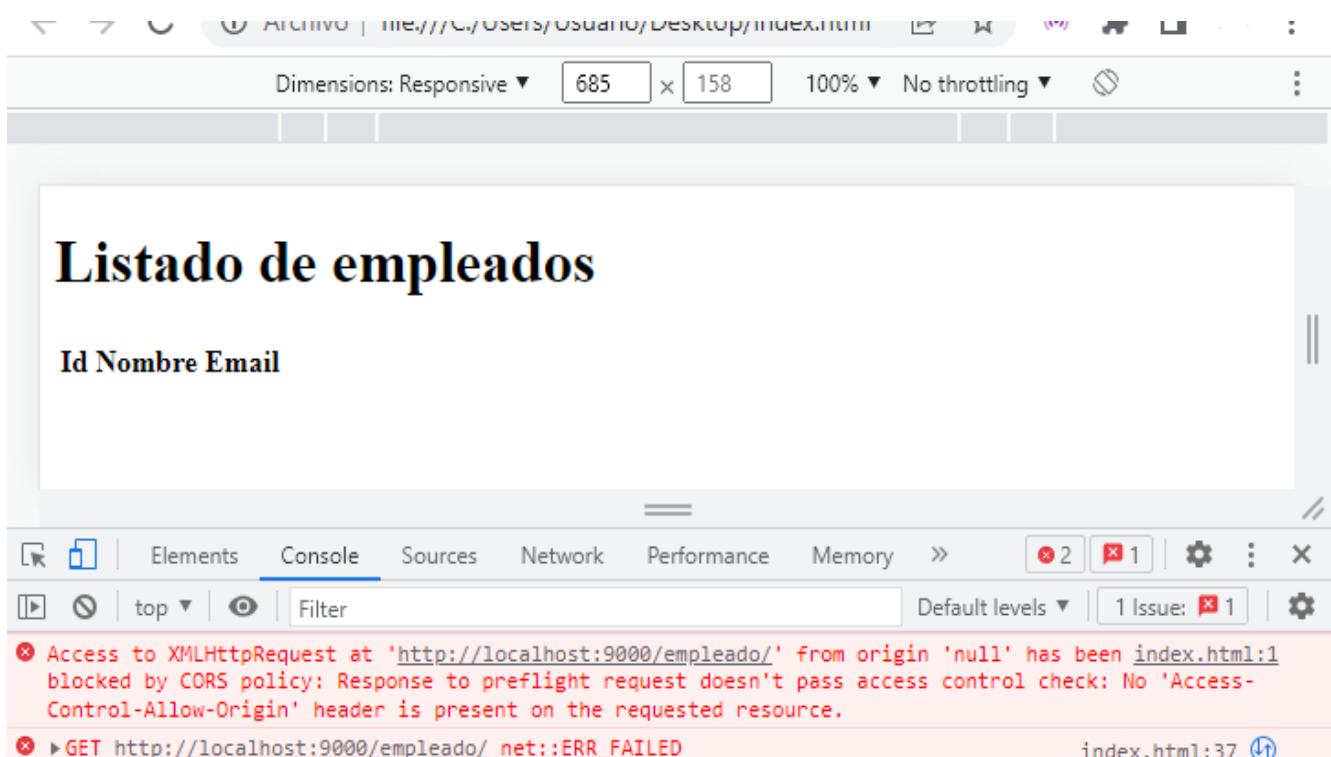
```

4.- **Axios**. A diferencia de las dos anteriores, es una librería externa que deberemos importar a nuestro proyecto para usarla. Axios era muy popular hasta la aparición de *fetch* por su sencillez y funcionalidades adicionales como establecer time outs en llamadas fallidas, y hacer conversiones JSON de forma automática en las respuestas.

Axios tiene más funcionalidades como los interceptores, que te permiten interceptar las llamadas y reaccionar a ellas, esto se usa por ejemplo cuando nuestro backend tiene un sistema de seguridad que necesita que las llamadas lleven un token, podemos meterle el token a la llamada desde un interceptor para no tener que picarlo en código cada vez que lo vayamos a usar

CORS

Si probamos el ejemplo anterior, desde fuera del servidor, obtendremos un error de seguridad, de tipo CORS y no podremos ejecutarlo correctamente.



El Intercambio de Recursos de Origen Cruzado (CORS) es un mecanismo que utiliza cabeceras HTTP adicionales para permitir que un cliente obtenga permiso para acceder a recursos seleccionados desde un servidor, en un origen distinto (dominio). En el ejemplo anterior ocurre esto, ya que la petición desde Javascript no está ubicada en *localhost*.

Por razones de seguridad, las aplicaciones Spring Boot restringen las solicitudes HTTP de origen cruzado iniciadas dentro de un script externo. Si ubicásemos la aplicación cliente (archivos .html, .js, etc.) en la carpeta */resources/static* no tendríamos este problema.

Para evitar este error debemos configurar el bean *WebMvcConfigurer* en el que podemos filtrar los orígenes permitidos a nivel verbo HTTP, cabeceras permitidas, y sobre todo los dominios origen permitidos: **una vez que nuestra aplicación cliente esté desplegada en un servidor, debería ser ese el único origen permitido**. Por simplificar, implementaremos una configuración global que permita cualquier tipo de acceso.

Obviamente esta configuración es menos segura pero más simple. Haríamos una clase anotada con `@Configuration` con el siguiente contenido.

```
@Configuration
public class CorsConfig {
    @Bean
    public WebMvcConfigurer corsConfigurer() {
        return new WebMvcConfigurer() {
            @Override
            public void addCorsMappings(CorsRegistry registry) {
                registry.addMapping("/**")
                    .allowedOrigins("*")
                    .allowedMethods("*")
                    .allowedHeaders("*");
            }
        };
    }
}
```

RestClient

Todo lo visto en este tema sobre API Rest hacía referencia a los recursos que nuestra aplicación servidor ponía a disposición de otros desarrolladores, bien clientes como navegadores o dispositivos móviles, bien otras aplicaciones de entorno servidor.

En este apartado vamos a ver el punto de vista opuesto, es decir, veremos cómo nuestra aplicación de servidor puede necesitar consumir APIs de otras aplicaciones servidor. Para esta tarea usaremos la interfaz `RestClient`. Obviamente los recursos que consumimos seguirán el estándar REST, pero no necesariamente estarán desarrolladas con Spring, para nuestra aplicación eso será indiferente.

RestClient es entonces una interfaz que representa el punto de entrada principal para hacer peticiones web. Es una novedad de Spring 6.1 que sustituye a WebClient de Spring 5, que a su vez sustituía la anterior RestTemplate. WebClient seguía los principios de programación reactiva y pertenecía a la librería `spring-webflux` por lo que había que añadir la dependencia `starter-webflux`, no siendo necesaria para la actual RestClient.

Para utilizarla simplemente, debemos crear una instancia de WebClient, mediante su método estático `create()`:

```
RestClient restClient = WebClient.create();
```

O bien:

```
String baseUri = "https://jsonplaceholder.typicode.com";
RestClient restClient = RestClient.create(baseUri);
```

Aunque también dispone del método `builder()`:

```
RestClient restClient = RestClient.builder()
    .baseUrl("https://jsonplaceholder.typicode.com")
    .defaultHeader(HttpHeaders.AUTHORIZATION, encodeBasic("username", "password"))
    .build();
```

Vamos a emplear una web que ofrece un servicio de API fake muy útil para pruebas. Su nombre es: <https://jsonplaceholder.typicode.com> y permite consultar una lista de tareas mediante la ruta `"/todos"`, consultar una tarea concreta mediante: `"/todos/{id}"`, añadir una nueva tarea mediante: `"/todos/"` con `post`, etc...

Luego, sobre esa instancia, podemos llamar a sus métodos: `get`, `post`, etc... que invocan los correspondientes verbos HTTP:

```
String result = restClient.get()
    .uri("/todos/3")
    .retrieve()
    .body(String.class);
System.out.println(result);
```

Si queremos obtener no solo el *body*, sino una respuesta completa, con cabeceras y códigos de estado, podemos añadir el método *toEntity* a la petición:

```
ResponseEntity<String> result = restClient.get()
    .uri("/todos/3")
    .retrieve()
    .toEntity(String.class);

System.out.println("Response status: " + result.getStatusCode());
System.out.println("Response headers: " + result.getHeaders());
System.out.println("Contents: " + result.getBody());
```

RestClient puede convertir también las respuestas JSON a objetos:

```
Todo todo = restClient.get()
    .uri("/todos/{id}", id)
    .accept(APPLICATION_JSON)
    .retrieve()
    .body(Todo.class);
```

Cuando lo que devuelve es una clase genérica, por ejemplo, *List* podemos usar la clase abstracta *ParameterizedTypeReference* para que el infiera el tipo real por nosotros. En el siguiente ejemplo, obtendríamos un *List* de *Todo*:

```
List<Todo> todos = restClient.get()
    .uri("/todos/")
    .accept(APPLICATION_JSON)
    .retrieve()
    .body(new ParameterizedTypeReference<>() {});
//.body(new ParameterizedTypeReference<List<Todo>>() {});
```

Si queremos hacer un POST o PUT, la petición es análoga:

```
Todo todo = new Todo(...);
ResponseEntity<Void> response = restClient.post()
    .uri("/todos/")
    .contentType(APPLICATION_JSON)
    .body(todo)
    .retrieve()
    .toBodilessEntity();
```

Y para borrado:

```
ResponseEntity<Void> response = restClient.delete()
    .uri("/todos/1")
    .retrieve().toBodilessEntity();
```

Gestión de errores

RestClient lanza una subclase de *RestClientException* cuando recibe un código de estado *4xx* o *5xx*, pero este comportamiento puede ser modificado por nosotros añadiendo *onStatus*:

```
String result = restClient.get()
    .uri("/wrong-url")
    .retrieve()
    .onStatus(HttpStatusCodes::is4xxClientError, (request, response) -> {
        throw new MyCustomRuntimeException(response.getStatusCode(), response.getHeaders())
    })
    .body(String.class);
```

Aplicación con RestClient

Vamos a ver un ejemplo completo de un servicio que realiza todas estas operaciones sobre una API de ejemplo. Este servicio podría inyectarse en cualquier otro servicio o en un controlador y emplearlo directamente.

Para poder llamar a los métodos de la API Rest necesitamos conocer por una parte las URLs que debemos invocar y por otra la estructura de los archivos JSON que recibiremos en caso de llamadas *get* o los que enviaremos en peticiones de tipo *post* o *put*.

Vamos a emplear una web que ofrece un servicio de API fake muy útil para pruebas. Su nombre es: <https://jsonplaceholder.typicode.com> y permite consultar una lista de tareas mediante la ruta "/todos", consultar una tarea concreta mediante: "/todos/{id}", añadir una nueva tarea mediante: "/todos/" con *post*, etc...

Podemos hacer una llamada de prueba desde Postman y ver la estructura del JSON devuelto:

The screenshot shows a Postman interface with a GET request to <https://jsonplaceholder.typicode.com/todos/3>. The response body is a JSON object:

```

1
2   "userId": 1,
3   "id": 3,
4   "title": "fugiat veniam minus",
5   "completed": false
6

```

Debemos entonces crear una clase que se corresponda con esa estructura (hay servicios en internet para crearla automáticamente desde el archivo JSON):

```

@Data
@NoArgsConstructor
@AllArgsConstructor
public class Todo {
    public int userId;
    public int id;
    public String title;
    public boolean completed;
}

```

Luego el servicio podría ser así:

```

@Service
public class RestClientService {
    private RestClient restClient;

    public RestClientService() {
        this.restClient = RestClient.create("https://jsonplaceholder.typicode.com");
    }
    public List<Todo> obtenerTodos() throws RuntimeException {
        String url = "/todos/";
        List<Todo> result = restClient.get()
            .uri(url)
            .retrieve()
            .body(new ParameterizedTypeReference<List<Todo>>() {
            });
        return result;
    }
    public Todo obtenerPorId(Integer id) throws RuntimeException {
        String url = "/todos/" + id;
        ResponseEntity<Todo> result = restClient.get()
            .uri(url)
            .retrieve()
            .toEntity(Todo.class);
        return result.getBody();
    }
}

```

```

public void añadir(Todo todo) throws RuntimeException {
    String url = "/todos/";
    ResponseEntity<Void> result = restClient.post()
        .uri(url)
        .contentType(MediaType.APPLICATION_JSON)
        .body(todo)
        .retrieve()
        .toBodilessEntity();
}
}

```

Ahora, solo deberíamos inyectar este servicio en cualquier componente (por ejemplo, un controlador u otro servicio) y llamar a sus métodos, teniendo en cuenta que puede lanzar excepciones hijas de *RestClientResponseException*. Ejemplo para obtener todas las 'todo'.

```

@Controller
public class RestClientController {

    @Autowired
    private RestClientService restClientService;

    @GetMapping("/lista")
    public String getAll(Model model) {
        try {
            model.addAttribute("todos", restClientService.obtenerTodos());
        } catch (RestClientResponseException e) {
            model.addAttribute("error", e.getMessage());
            return "errorView";
        }
        return "showAllView";
    }
}

```

Se podrían definir en el servicio métodos análogos para otras operaciones, como obtener una sola tarea, añadir una nueva tarea, etc., siempre que la API destino nos lo permita.

```

@GetMapping("/get/{id}")
public String getOne(@PathVariable Integer id, Model model) {
    try {
        model.addAttribute("todo", restClientService.obtenerPorId(id));
    } catch (RestClientResponseException e) {
        model.addAttribute("error", e.getMessage());
        return "errorView";
    }
    return "showOneView";
}

@PostMapping("/nuevo")
public String postOne(Todo todo, Model model) {
    try {
        restClientService.añadir(todo);
        model.addAttribute("todo", todo);
    } catch (RestClientResponseException e) {
        model.addAttribute("error", e.getMessage());
        return "errorView";
    }
    return "showPostView";
}

```

Seguridad en API REST

Como comentamos al principio este tema, el modelo de sesión con cookies es válido para web orientadas a navegadores, como las que desarrollamos con Spring MVC. Cuando hablamos de aplicaciones API REST debemos ser capaces de dar respuesta a distintos tipos de clientes y muchos de ellos no trabajarán con cookies. Para este tipo de aplicaciones existen otros métodos como JWT o OAuth2. En este manual veremos la primera opción.

JSON Web Token: JWT

Cuando un usuario se identifica, en vez de crear una sesión, el servidor genera un *token* y se lo envía al cliente y éste, en las sucesivas peticiones que haga al servidor, adjuntará dicho *token*. Para almacenar el token en el cliente, dependerá del tipo de dispositivo y aplicación.

Generalmente el token se envía en la cabecera de la petición. Al llegar entonces cualquier petición el servidor validará que el token recibido corresponde con el enviado inicialmente y procederá a enviar la respuesta. El token no securiza los datos, no los encripta, solo garantiza el origen de los datos, ante un ataque de tipo *Man-in-the-middle* se podría obtener información del usuario, para evitar esto deberíamos emplear encriptado HTTPS.



No vamos a entrar en detalle, pero diremos en un JWT se compone de tres partes: **header** (con la identificación y algoritmo empleado para generar el token), **payload** (con el contenido del token, formado por datos propios como la fecha de expiración del token y datos del usuario: id, nombre, contraseña, etc.) y **signature** (firma o *hash* obtenido al codificar el token).

Configuración JWT en el proyecto API Rest

Para securizar una aplicación API Rest mediante JWT debemos seguir varios pasos, algunos comunes a los vistos para aplicaciones Spring MVC con sesión/cookies pero también con algunos nuevos, referentes a la construcción y validación del token JWT, que no emplea cookies.

Vamos a describirlos paso a paso, sobre un proyecto genérico con varios perfiles (roles) que dispondrán de diversos niveles de acceso a los endpoints del API.

En el archivo de recursos proporcionado con este manual se entrega el código fuente del proyecto que se describe a continuación y que puede servir de base para la realización de cualquier otro. Suponemos que tenemos creada una entidad Usuario:

```

@Data
@AllArgsConstructor
@NoArgsConstructor
@EqualsAndHashCode(of = "id")
@Entity
@Table(name = "users", uniqueConstraints = {
    @UniqueConstraint(columnNames = "nombre"),
    @UniqueConstraint(columnNames = "email")
})
public class Usuario {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;

    @NotBlank @Size(max = 20)
    private String nombre;

    @NotBlank @Size(max = 50) @Email
    private String email;

    @NotBlank
    private String password;

    private Rol rol;
}

```

Y su repositorio:

```

public interface UsuarioRepository extends JpaRepository<Usuario, Long> {
    Usuario findByNombre(String nombre);
    Boolean existsByNombre(String nombre);
    Boolean existsByEmail(String email);
}

```

Estos serían los pasos a seguir, suponiendo que ya tenemos una entidad Usuario, con atributos: id, y

1.- Incorporar la dependencia *starter-security* en el archivo *pom.xml*, al igual que en Spring MVC y adicionalmente, tres dependencias *jsonwebtoken*:

```

<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-security</artifactId>
</dependency>
<dependency>
    <groupId>io.jsonwebtoken</groupId>
    <artifactId>jjwt-api</artifactId>
    <version>0.11.5</version>
</dependency>
<dependency>
    <groupId>io.jsonwebtoken</groupId>
    <artifactId>jjwt-impl</artifactId>
    <version>0.11.5</version>
    <scope>runtime</scope>
</dependency>
<dependency>
    <groupId>io.jsonwebtoken</groupId>
    <artifactId>jjwt-jackson</artifactId>
    <version>0.11.5</version>
    <scope>runtime</scope>
</dependency>

```

2.- Crear la clase SecurityConfig (similar a la creada en proyectos Spring MVC, pero con alguna diferencia) en la que se crean todos los beans necesarios:

```

@Configuration
@EnableMethodSecurity
public class SecurityConfig {

    @Autowired private UserDetailsService userDetailsService;

    @Autowired private AuthEntryPointJwt authEntryPointJwt;

    @Bean
    public AuthTokenFilter authenticationJwtTokenFilter() {
        return new AuthTokenFilter();
    }

    @Bean
    public DaoAuthenticationProvider authenticationProvider() {
        DaoAuthenticationProvider authProvider = new DaoAuthenticationProvider();
        authProvider.setUserDetailsService(userDetailsService);
        authProvider.setPasswordEncoder(passwordEncoder());
        return authProvider;
    }

    @Bean
    public AuthenticationManager authenticationManager
            (AuthenticationConfiguration authConfig) throws Exception {
        return authConfig.getAuthenticationManager();
    }

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

    @Bean
    public SecurityFilterChain filterChain(HttpSecurity http) throws Exception {
        http.csrf(csrf -> csrf.disable())
            .exceptionHandling(exception ->
                exception.authenticationEntryPoint(authEntryPointJwt))
            .sessionManagement(session ->
                session.sessionCreationPolicy(SessionCreationPolicy.STATELESS))
            .authorizeHttpRequests(auth -> auth
                .requestMatchers("/api/auth/**", "/error").permitAll()
                .requestMatchers("/api/test/all").permitAll()
                .requestMatchers("/api/test/user").hasAnyRole("USER", "MANAGER", "ADMIN")
                .requestMatchers("/api/test/manager").hasAnyRole("MANAGER", "ADMIN")
                .requestMatchers("/api/test/admin").hasAnyRole("ADMIN"));
        http.authenticationProvider(authenticationProvider());
        http.addFilterBefore(authenticationJwtTokenFilter(),
            UsernamePasswordAuthenticationFilter.class);
        http.cors(Customizer.withDefaults());
        return http.build();
    }
}

```

Recordar lo que comentamos en apartados anteriores que `requestMatchers` admite también se le pase el verbo HTTP sobre el que queremos trabajar:

```
.requestMatchers(HttpMethod.POST, "/path").denyAll()
```

3.- Crear las clases que implementan `UserDetailsService` y `UserDetails` que gestionan el usuario que está conectado en este momento. La primera clase accede al repositorio de usuarios de la aplicación. Para simplificarlo, vamos a considerar que cada usuario tiene un solo rol.

UserDetailsServiceimpl.java:

```
@Service
public class UserDetailsServiceImpl implements UserDetailsService {
    @Autowired
    private UsuarioRepository usuarioRepository;

    @Override
    @Transactional
    public UserDetails loadUserByUsername(String nombre) throws UsernameNotFoundException {
        Usuario usuario = usuarioRepository.findByNombre(nombre);
        if (usuario==null) throw new UsernameNotFoundException("Usuario no encontrado");
        return UserDetailsImpl.build(usuario);    //ojo: es distinto a la seguridad en MVC !!
    }
}
```

UserDetailsImpl.java (*esta clase tampoco era necesaria en aplicaciones MVC, no API Rest*)

```
@Data
@AllArgsConstructor
@NoArgsConstructor
@EqualsAndHashCode(of = "id")

public class UserDetailsImpl implements UserDetails {
    private Long id;
    private String username;
    private String email;

    @JsonIgnore
    private String password;

    private Collection<? extends GrantedAuthority> authorities;

    public static UserDetailsImpl build(Usuario usuario) {
        List<GrantedAuthority> authorities = new ArrayList<>();
        authorities.add(new SimpleGrantedAuthority("ROLE_" + usuario.getRol().name()));
        return new UserDetailsImpl(
            usuario.getId(),
            usuario.getNombre(),
            usuario.getEmail(),
            usuario.getPassword(),
            authorities);
    }
    @Override
    public boolean isAccountNonExpired() { return true; }

    @Override
    public boolean isAccountNonLocked() { return true; }

    @Override
    public boolean isCredentialsNonExpired() { return true; }

    @Override
    public boolean isEnabled() { return true; }
}
```

4.- Debemos crear las clases **AuthEntryPointJwt**, **AuthTokenFilter** y **JwtUtils** para la gestión del token en la que fijamos todos sus parámetros. No se incluye aquí el código ya que no nos aportan ningún concepto adicional, pero sí en el archivo de recursos disponible en el área de descargas.

5.- La aplicación tendrá tres clases *dto* para la gestión de los usuarios:

a) **LoginDto** : con los datos del usuario que se identifica en la aplicación:

```
@Getter  
@Setter  
public class LoginDto {  
  
    @NotBlank  
    private String nombre;  
  
    @NotBlank  
    private String password;  
}
```

b) **SignupDto** : con los datos de registro de usuario.

```
@Getter  
@Setter  
public class SignupDto {  
    @NotBlank  
    @Size(min = 3, max = 20)  
    private String nombre;  
  
    @NotBlank  
    @Size(min = 6, max = 40)  
    private String password;  
  
    @NotBlank  
    @Size(max = 50)  
    @Email  
    private String email;  
  
    private String rol;  
}
```

c) **JwtResponseDto** : son los datos que se envían al cliente cuando se ha validado correctamente el usuario y contraseña y se ha permitido el acceso.

```
@Data  
@AllArgsConstructor  
public class UsuarioJwt {  
    private String accessToken;  
    private String tokenType;  
    private Long id;  
    private String nombre;  
    private String email;  
    private String rol;  
}
```

Hay que fijarse en el primer atributo de esta clase, además de todos los datos del usuario, se envía el token que tendrá que ser añadido a todas las peticiones siguientes.

6.- Los roles pueden ser entidades con su propio repositorio implementando un CRUD sobre el mismo o, de forma más sencilla, una simple enumeración:

```
public enum Rol { USER, MANAGER, ADMIN }
```

7.- Debemos crear el controlador que se encargará tanto del registro de usuarios (`/signup`) como del login de los mismos (`/signin`). **No crearemos controlador para el logout, ya que la sesión finaliza al expirar el token.**

```

@CrossOrigin(origins = "*", maxAge = 3600)
@RestController
@RequestMapping("/api/auth")
public class AuthController {
    @Autowired
    private AuthenticationManager authenticationManager;

    @Autowired
    private UsuarioRepository usuarioRepository;

    @Autowired
    private PasswordEncoder encoder;

    @Autowired
    private JwtUtils jwtUtils;

    @PostMapping("/signin")
    public ResponseEntity<?> authenticateUser(@Valid @RequestBody LoginDto loginDto) {
        Authentication authentication = authenticationManager.authenticate(
            new UsernamePasswordAuthenticationToken(loginDto.getNombre(),
                loginDto.getPassword()));
        SecurityContextHolder.getContext().setAuthentication(authentication);
        String jwt = jwtUtils.generateJwtToken(authentication);

        UserDetailsImpl userDetails = (UserDetailsImpl) authentication.getPrincipal();
        String rol = userDetails.getAuthorities().stream()
            .findFirst().map(a -> a.getAuthority()).orElse("ERROR");
        return ResponseEntity.ok(new JwtResponseDto(jwt, "Bearer",
            userDetails.getId(),
            userDetails.getUsername(),
            userDetails.getEmail(),
            rol));
    }

    @PostMapping("/signup")
    public ResponseEntity<?> registerUser(@Valid @RequestBody SignupDto signUpRequest) {
        if (usuarioRepository.existsByNombre(signUpRequest.getNombre())) {
            return ResponseEntity
                .badRequest()
                .body(new MessageResponse("Error: Ya existe un usuario con ese nombre"));
        }

        if (usuarioRepository.existsByEmail(signUpRequest.getEmail())) {
            return ResponseEntity
                .badRequest()
                .body(new MessageResponse("Error: Ya existe un usuario con ese email"));
        }

        // Create new user's account
        Usuario usuario = new Usuario(null, signUpRequest.getNombre(),
            signUpRequest.getEmail(),
            encoder.encode(signUpRequest.getPassword()),
            Rol.valueOf(signUpRequest.getRol()));
        usuarioRepository.save(usuario);
        return ResponseEntity.ok(new MessageResponse("Usuario registrado correctamente"));
    }
}

```

8.- Habría que añadir a la aplicación los controladores y servicios con la lógica de negocio. Un ejemplo de prueba podría ser este:

```
@CrossOrigin(origins = "*", maxAge = 3600)
@RestController
@RequestMapping("/api/test")
public class TestController {
    @GetMapping("/all")
    public String showContentForAll() {
        return "Contenido público";
    }

    @GetMapping("/user")
    public String showContentForUsers() {
        return "Contenido para usuarios";
    }

    @GetMapping("/manager")
    public String showContentForManager() {
        return "Contenido para usuarios de tipo Manager";
    }

    @GetMapping("/admin")
    public String showContentForAdmins() {
        return "Contenido para administradores";
    }
}
```

9.- Para completar el proyecto necesitaríamos la clase *User* y su repositorio *UserRepository*, la clase *MessageResponse*, y la configuración de CORS.

```
@Getter
@Setter
@AllArgsConstructor
public class MessageResponse {
    private String message;
}
```

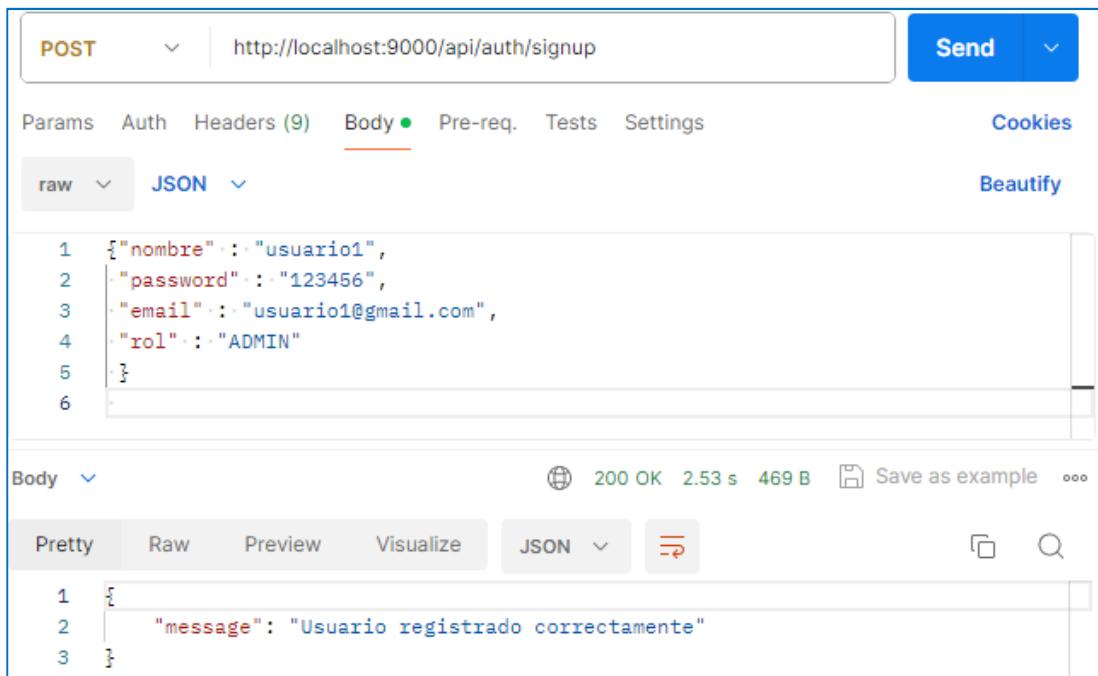
En temas anteriores ya habíamos hablado de la configuración CORS, pero permitíamos cualquier tipo de tráfico. En este momento aún no tenemos desplegada la aplicación cliente, por lo que no podemos limitar el “origin” de las peticiones, pero si podemos restringir los verbos HTTP permitidos y el tipo de peticiones:

```
@Configuration
public class CorsConfig {
    @Bean
    public WebMvcConfigurer corsConfigurer() {
        return new WebMvcConfigurer() {
            @Override
            public void addCorsMappings(CorsRegistry registry) {
                registry.addMapping("/**")
                    .allowedOrigins("*") //dominio de la app cliente
                    .allowedMethods("GET", "POST", "PUT", "DELETE")
                    .allowedHeaders("Content-Type", "Authorization", "Origin", "Accept");
            }
        };
    }
}
```

Testing de la autenticación JWT

Vamos a probar el proyecto completo desde Postman (está disponible en el área de descargas).

1.- Registrar un nuevo usuario, usando el DTO correspondiente.



POST http://localhost:9000/api/auth/signup

Body (JSON)

```

1 {"nombre": "usuario1",
2 "password": "123456",
3 "email": "usuario1@gmail.com",
4 "rol": "ADMIN"
5 }
6

```

Body

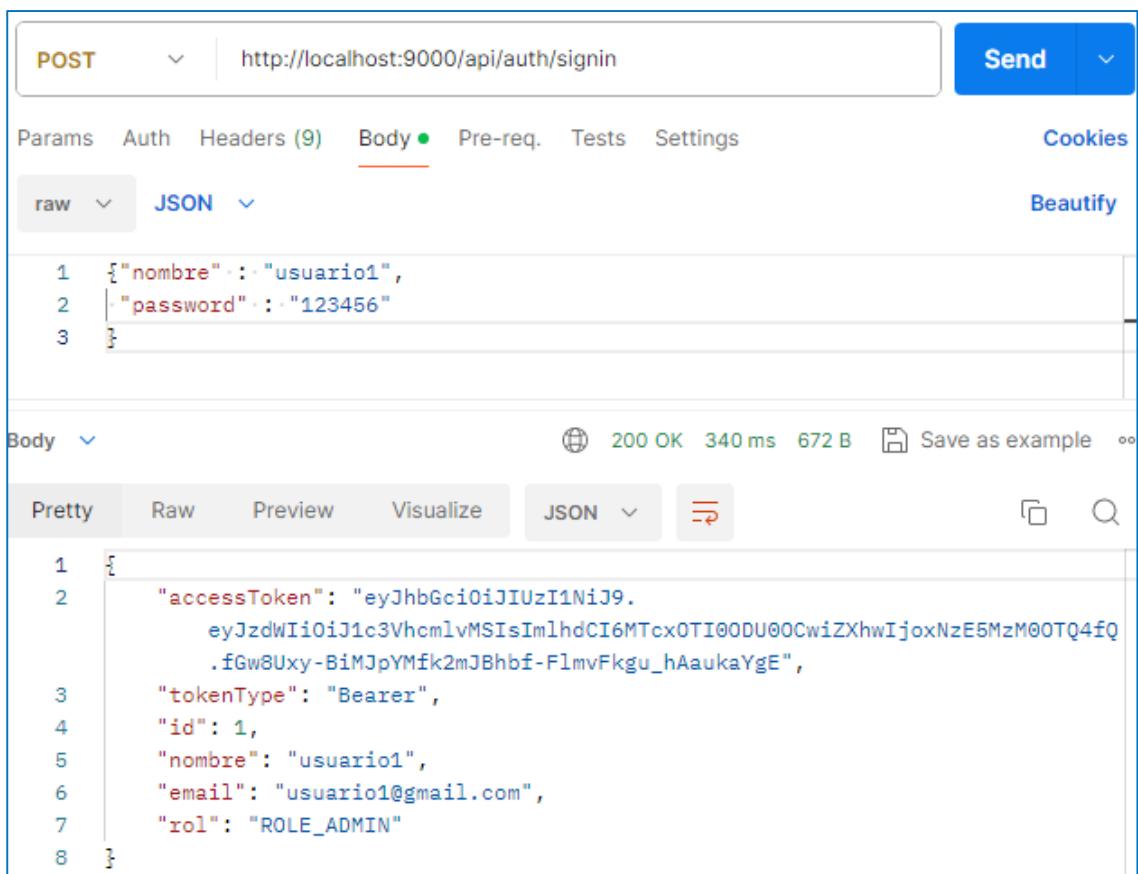
200 OK 2.53 s 469 B

```

1 {
2   "message": "Usuario registrado correctamente"
3 }

```

2.- Hacer login con ese usuario. Importante: devuelve el token asignado y debemos incluirlo en las siguientes peticiones.



POST http://localhost:9000/api/auth/signin

Body (JSON)

```

1 {"nombre": "usuario1",
2 "password": "123456"
3 }

```

Body

200 OK 340 ms 672 B

```

1 {
2   "accessToken": "eyJhbGciOiJIUzI1NiJ9.eyJzdWIiOiJ1c3Vhcm1vMSIsImhdCI6MTcxOTI0ODU0OCwiZXhwIjoxNzE5MzM0OTQ4fQ.fGw8Uxy-BiMJpYMFk2mJBhbf-F1mvFkgu_hAaukaYgE",
3   "tokenType": "Bearer",
4   "id": 1,
5   "nombre": "usuario1",
6   "email": "usuario1@gmail.com",
7   "rol": "ROLE_ADMIN"
8 }

```

3.- Hacer petición a zona restringida, añadiendo en el campo “Authorization” el token en la cabecera de la petición, precedida de la palabra “Bearer” y un espacio en blanco.

The screenshot shows the Postman interface with a GET request to `http://localhost:9000/api/test/admin`. The 'Headers' tab is active, displaying the following table:

Key	Value	Description	...	Bulk Edit	Presets
Authorization	Bearer eyJhbGciOiJIUzI1NiJ9....				
Key	Value	Description			

Below the table, the response status is 200 OK (21 ms, 453 B). The response body contains the text: `1 Contenido para administradores`.

Refresh del token

Una última tarea relacionada con el token sería su regeneración (“refresh”) en caso de que haya expirado para que el cliente pueda seguir trabajando sin tener que enviar sus credenciales de nuevo. Dejamos el proceso en el siguiente enlace:

<https://www.bezkoder.com/spring-boot-refresh-token-jwt/#Refresh Token Request and Response>

Acceso desde el cliente con autenticación JWT

Mostramos, a modo de ejemplo, como se consumiría desde JavaScript una API con una autenticación previa mediante JWT. Emplearemos la técnica de *fetch* con *async/await*.

El primer paso sería la función de autenticación que recibiría un usuario y contraseña y devolvería el token JWT:

```
async function api_signin(username, pass) {
  const credenciales = { username: username, password: pass };
  try {
    const response = await fetch("localhost:9000/api/auth/signin", {
      method: 'POST',
      headers: {
        'Content-Type': 'application/json',
      },
      body: JSON.stringify(credenciales),
    });
    if (!response.ok) { const textErr = await response.text(); throw new Error(textErr); }
    const { accessToken } = await response.json();
    return accessToken;
  } catch (error) { console.error('Error al obtener datos:', error); return false; }
}
```

Una vez recibido el token, éste deberá ser guardado (bien como variable global en el script, bien en LocalStorage, etc.) e incorporarlo en las siguientes peticiones al servidor. Ponemos como ejemplo lo que sería una modificación de un recurso (PUT) de tipo *empleado* a través de su *id*.

```

async function api_put(id, data, token) {
  try {
    const response = await fetch(`localhost:9000/empleado/${id}`, {
      method: 'PUT',
      headers: {'Content-Type': 'application/json',
                'Authorization': `Bearer ${token}`},
      body: JSON.stringify(data)
    });
    if (!response.ok) { const textErr = await response.text(); throw new Error(textErr); }
    return true;
  } catch (error) {console.error('Error al modificar datos:', error); return false; }
}

```

Seguridad en Aplicación mixta con MVC y API REST

En este capítulo hemos visto como securizar una aplicación API REST mediante JWT y en el tema anterior vimos como hacer lo mismo con una aplicación web tipo MVC basándonos en sesiones de navegador. La duda que nos puede surgir ahora es cómo llevar a cabo este proceso de securización en una aplicación que tenga una parte MVC con vistas y Thymeleaf y otra parte con endpoints API REST.

El proceso a seguir sería el siguiente:

- 1) Las URL de la parte MVC y API deben estar claramente delimitadas. Una forma sencilla de hacerlo es que todas las rutas de la API comiencen por `/api`.
- 2) Tanto para una parte como para la otra debemos seguir todos los pasos tal cual se han explicado en ambos capítulos.
- 3) Implementando el paso anterior, nos encontraremos una diferencia en la clase `UserDetailImpl` ya que no se define de la misma forma en el caso de MVC y en el de API. Debemos usar el código explicado en este tema de API, que será válido para ambos esquemas.
- 4) Una de las diferencias más notables entre ambos esquemas es el contenido del bean `SecurityFilterChain` definido en la clase `SecurityConfig`. La solución para estas aplicaciones mixtas será crear dos beans diferentes, indicando sobre el referente a API REST las URLs a las que afecta, que serán las que comiencen por `/api` según comentamos en el primer punto de este apartado. La clase `SecurityConfig` quedaría así:

```

@Configuration
@EnableWebSecurity
@EnableMethodSecurity
public class SecurityConfig {
    @Autowired private UserDetailsService userDetailsService;
    @Autowired private AuthEntryPointJwt authEntryPointJwt;
    @Bean
    public AuthTokenFilter authenticationJwtTokenFilter() {
        return new AuthTokenFilter();
    }
    @Bean
    public DaoAuthenticationProvider authenticationProvider() {
        DaoAuthenticationProvider authProvider = new DaoAuthenticationProvider();
        authProvider.setUserDetailsService(userDetailsService);
        authProvider.setPasswordEncoder(passwordEncoder());
        return authProvider;
    }
    @Bean
    public AuthenticationManager authenticationManager(
        AuthenticationConfiguration authenticationConfiguration) throws Exception {
        return authenticationConfiguration.getAuthenticationManager();
    }
    @Bean
    public PasswordEncoder passwordEncoder() { return new BCryptPasswordEncoder();
}

```

```
@Bean
@Order(1)
public SecurityFilterChain ApifilterChain(HttpSecurity http) throws Exception {
    http.securityMatcher("/api/**");
    http
        .csrf(csrf -> csrf.disable())
        .exceptionHandling(exception -> exception.authenticationEntryPoint(authEntryPointJwt))
        .sessionManagement(session -> session)
        .sessionCreationPolicy(SessionCreationPolicy.STATELESS)
        .authorizeHttpRequests(auth -> auth
            // PERMISOS API
            .requestMatchers("/api/**").permitAll()      //añadir aquí todos los permisos api
        );
    http.authenticationProvider(authenticationProvider());
    http.addFilterBefore(authenticationJwtTokenFilter(),
        UsernamePasswordAuthenticationFilter.class);
    http.cors(Customizer.withDefaults());
    return http.build();
}

@Bean
@Order(2)
public SecurityFilterChain MvcfilterChain(HttpSecurity http) throws Exception {
    http.headers(headersConfigurer -> headersConfigurer
        .frameOptions(HeadersConfigurer.FrameOptionsConfig::sameOrigin));
    http
        .authorizeHttpRequests(auth -> auth
    // PERMISOS MVC
        .requestMatchers("/**").permitAll()      //añadir aquí todos los permisos mvc
        .formLogin(httpSecurityFormLoginConfigurer -> httpSecurityFormLoginConfigurer
            .LoginPage("/signin")
            .loginProcessingUrl("/login")
            .failureUrl("/signin?error")
            .defaultSuccessUrl("/public/home", true).permitAll())
        .logout(logout) -> logout
            .logoutSuccessUrl("/public/home").permitAll()
        .csrf(csrf -> csrf.ignoringRequestMatchers(PathRequest.toH2Console()))
        .rememberMe(Customizer.withDefaults())
        .httpBasic(Customizer.withDefaults())
        .exceptionHandling(exceptions -> exceptions.accessDeniedPage("/accessError")));
    return http.build();
}
}
```

Tema 10: Pase a producción y Testing

Introducción

Una vez terminado el desarrollo de una aplicación, quedan aún tareas importantes a realizar como puede ser el testing (JUnit, Mockito) y documentación (JavaDoc, Swagger), aunque idealmente deberían hacerse a la par del desarrollo; y por último el despliegue en un entorno de producción.

La aplicación desarrollada puede ser empaquetada de dos formas diferentes, en un archivo ".jar" que contendrá su propio servidor web, o bien en un archivo ".war" que necesitará un servidor web como veremos a continuación.

En esta fase de despliegue podríamos hablar de la **integración continua**, esto es, la automatización del pase a producción, con todo lo que ello conlleva: compilación, ejecución de los test automatizados, integración con el resto de las aplicaciones, etc. Existen herramientas para esta tarea como **Jenkins**.

Otro aspecto importante es el uso de **contenedores** con tecnologías como **Docker** o **Kubernetes**, etc. Un contenedor vendría a ser una máquina virtual que contendría nuestro sistema operativo, el servidor web, la aplicación, todas las librerías necesarias y entornos de ejecución, configuración, etc. para que la aplicación se pueda ejecutar sobre cualquier máquina real. En definitiva, los contenedores representan un mecanismo de empaquetado lógico donde las aplicaciones tienen todo lo que necesitan para ejecutarse.

En este tema hablaremos de diferentes aspectos a incorporar en nuestras aplicaciones para su pase a producción.

Empaquetado “war”

Hasta ahora hemos empaquetado nuestras aplicaciones, en un archivo ".jar" que contenía su propio servidor web integrado y por lo tanto era autónomo para su ejecución. Otra posibilidad es empaquetar en un archivo ".war" que necesitará un servidor web en el que desplegarse, compatible con la tecnología empleada, en nuestro caso para Spring; podemos usar Tomcat, GlassFish, JBoss, etc.

Los pasos a seguir serían los siguientes:

- Al generar la aplicación (desde el IDE o start.spring.io) indicar que el empaquetado es “war”. Esto generará los siguientes cambios en nuestro proyecto, con respecto al empaquetado .jar.
 - Añadirá en el archivo de configuración *pom.xml*/la línea: `<packaging>war</packaging>`
 - Añadirá en el *pom.xml*/dependencia *starter-tomcat*.
 - Creará una clase adicional llamada *ServletInitializer* con el siguiente código:

```
public class ServletInitializer extends SpringBootServletInitializer {
    @Override
    protected SpringApplicationBuilder configure(SpringApplicationBuilder application) {
        return application.sources(DemoApplication.class);
    }
}
```

- En la terminal de Visual Studio Code, en la raíz del proyecto, ejecutamos el comando Maven:

`mvn clean package`

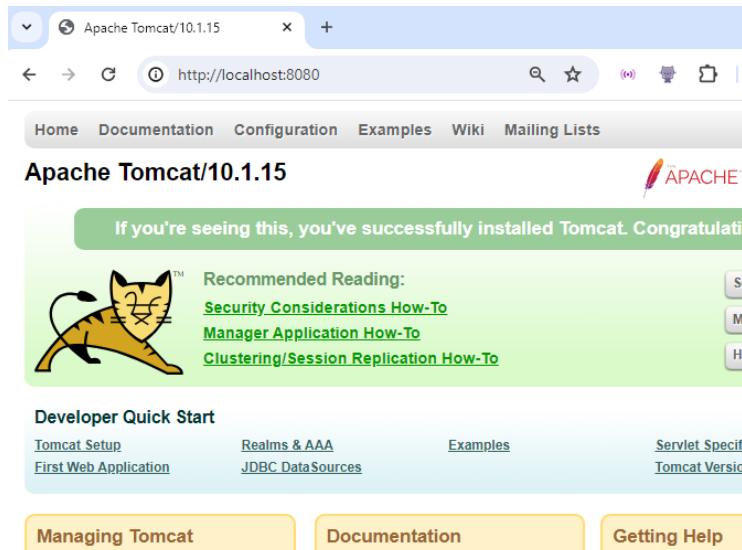
y obtendremos el “ejecutable” de nuestra aplicación en formato *.war* en la carpeta *target*. El nombre del archivo estará formado por el *artifactId* y *versión* establecidos en el *pom.xml*. Podríamos obtenerlo con otro nombre añadiendo en el *pom.xml*/la etiqueta:

`<finalName>myAppName</finalName>`

- El siguiente paso será instalar Apache Tomcat: lo descargamos desde <https://tomcat.apache.org/>, descomprimimos en una carpeta cualquiera y añadimos la variable de entorno CATALINA_HOME con la ruta a esa carpeta en la que descomprimimos. También necesita la variable JAVA_HOME apuntando a la ruta del *jdk*. El puerto por defecto de Tomcat es 8080, por lo que, si no lo cambiamos, la aplicación Spring debería tener el mismo puerto.

Hay que instalar Tomcat 10 o superior para proyectos Spring 3 (usan jakarta en vez de javax)

- Arrancamos el servidor Tomcat ejecutando: *apache-tomcat-10\bin\startup.bat*. Podemos comprobar su funcionamiento visitando: <http://localhost:8080>



- Movemos el archivo *.war* a la carpeta *apache-tomcat-10\webapps* y pasados unos segundos se generará una carpeta con el mismo nombre que el archivo *.war* y la aplicación quedará instalada.
- Ahora ya podríamos eliminar el archivo *.war* ya que la aplicación realmente está en la carpeta que acabamos de obtener. Para acceder a la aplicación: <http://localhost:8080/nombreCarpeta>
- A diferencia de lo que ocurre en el empaquetado *.jar*, los archivos de la aplicación no están instalados en la raíz de servidor, sino que están en una carpeta con el nombre del proyecto. Es aquí donde podemos comprobar la diferencia entre emplear en una página *html* con *Thymeleaf* la etiqueta: `` y la etiqueta: `<a th:href="@{/}">` La primera nos llevaría a la raíz del servidor, esto es, a la página que se muestra en la imagen anterior, y la segunda nos llevaría a la raíz de nuestra aplicación.
- Por último, para parar el servidor Tomcat: *apache-tomcat-10\bin\shutdown.bat*.

MySQL

MySQL es un gestor de bases de datos relacional bajo licencia dual (libre y comercial), perteneciente a Oracle y una de las más populares actualmente, empleada por Wikipedia, Twitter, etc. Existe un *fork* llamado MariaDB, desarrollado por el creador de MySQL, y con licencia GPL.

Instalación

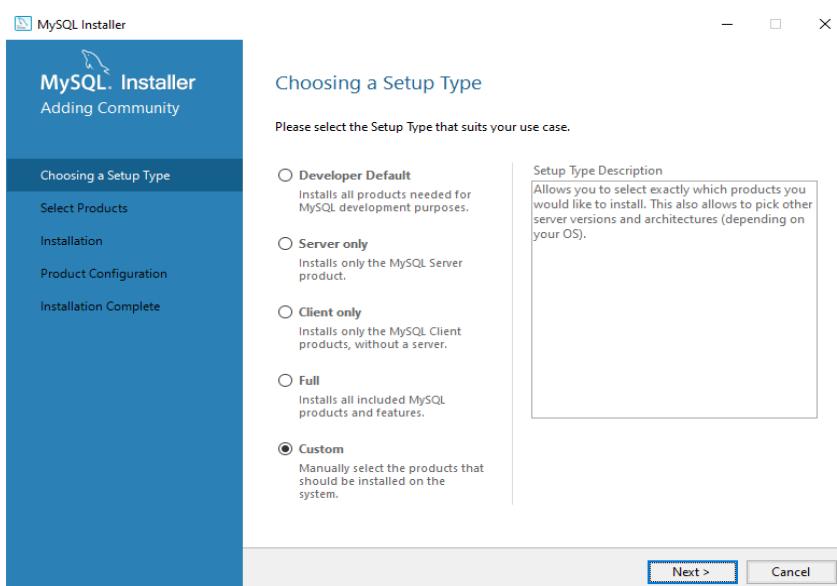
Lo primero que debemos hacer es descargar el archivo de instalación del servidor (MySQL versión Community) desde: <https://dev.mysql.com/downloads/mysql/>

The screenshot shows the MySQL Community Downloads page for MySQL Server 8.0.29. The 'General Availability (GA) Releases' tab is selected. A dropdown menu shows 'Microsoft Windows' as the selected operating system. The 'Recommended Download' section features the 'MySQL Installer for Windows' which is described as 'All MySQL Products. For All Windows Platforms. In One Package.' Below it, there are links for 'Windows (x86, 32 & 64-bit), MySQL Installer MSI' and 'Windows (x86, 64-bit), ZIP Archive'. Other download options like 'Windows (x86, 64-bit), ZIP Archive' and 'Debug Binaries & Test Suite' are also listed with their respective file sizes and download links.

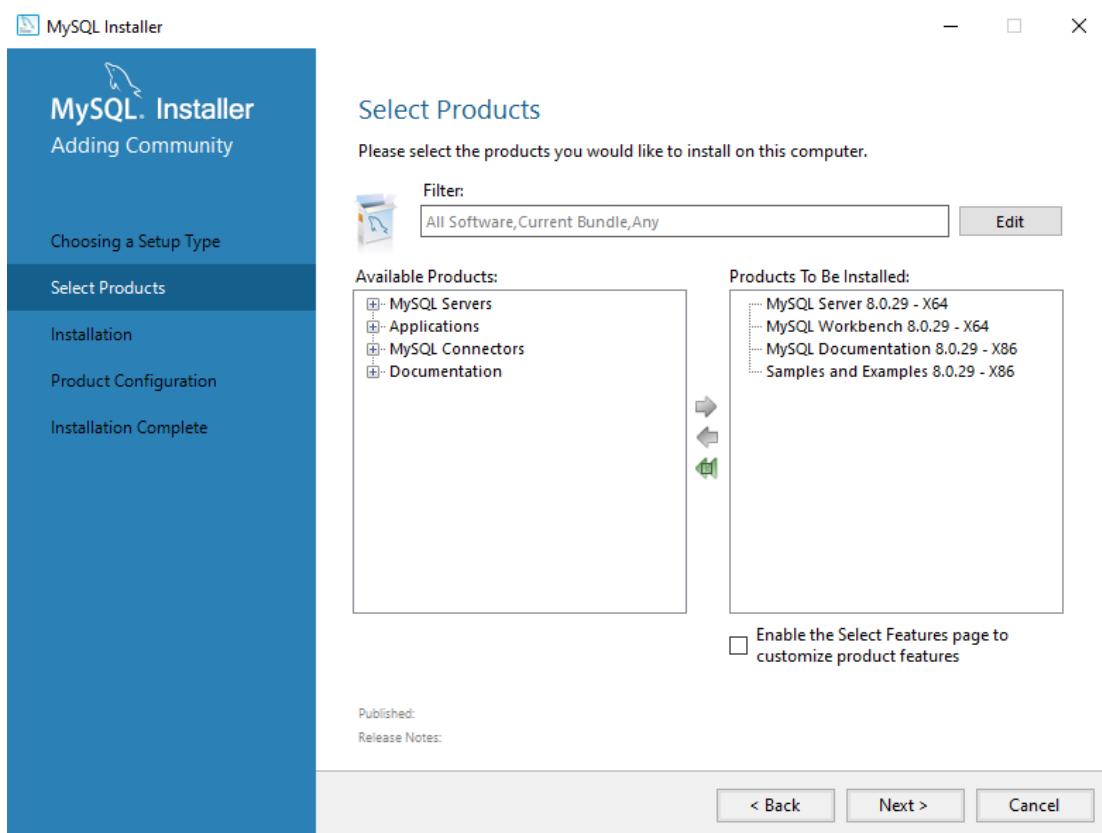
Elegimos “*MySQL Installer for Windows*” (*All MySQL Products*), que incluirá también MySQL Workbench, herramienta que usaremos para conectarnos al servidor para tareas administrativas como arrancar y parar el servidor y desde la que podremos realizar consultas a la base de datos.

Instalar MySQL Workbench no es obligatorio ya que podríamos acceder a MySQL mediante otras herramientas como puede ser `phpmyadmin`.

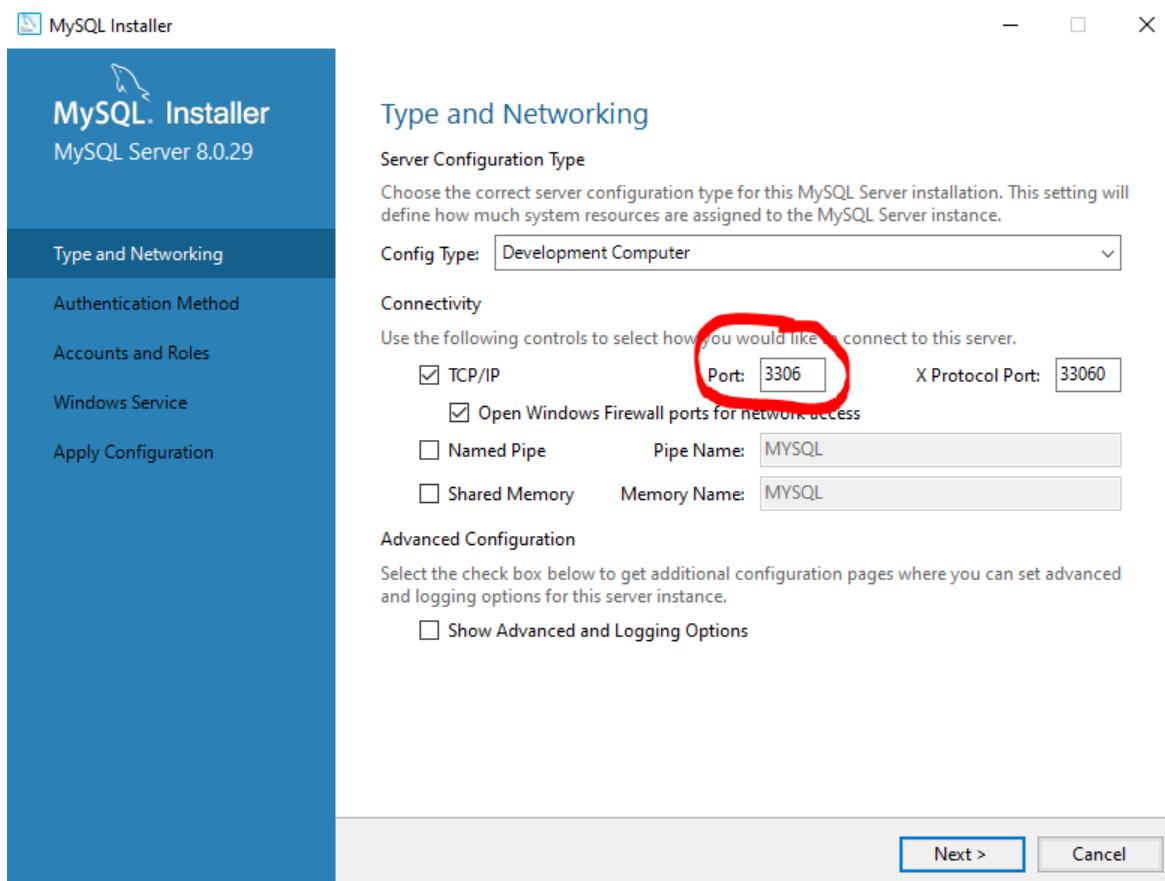
Hacemos doble click sobre el instalador. En el proceso de instalación, seleccionamos el tipo de instalación “Custom”.



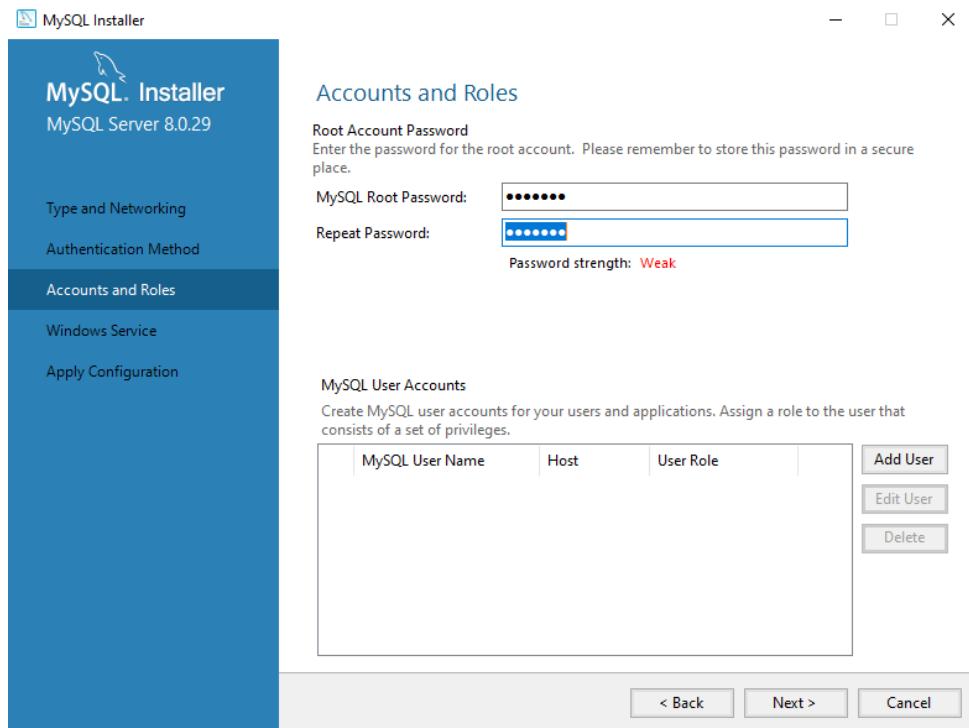
Y en los productos a instalar, seleccionamos solo el MySQL Server, MySQL WorkBench y documentación y ejemplos.



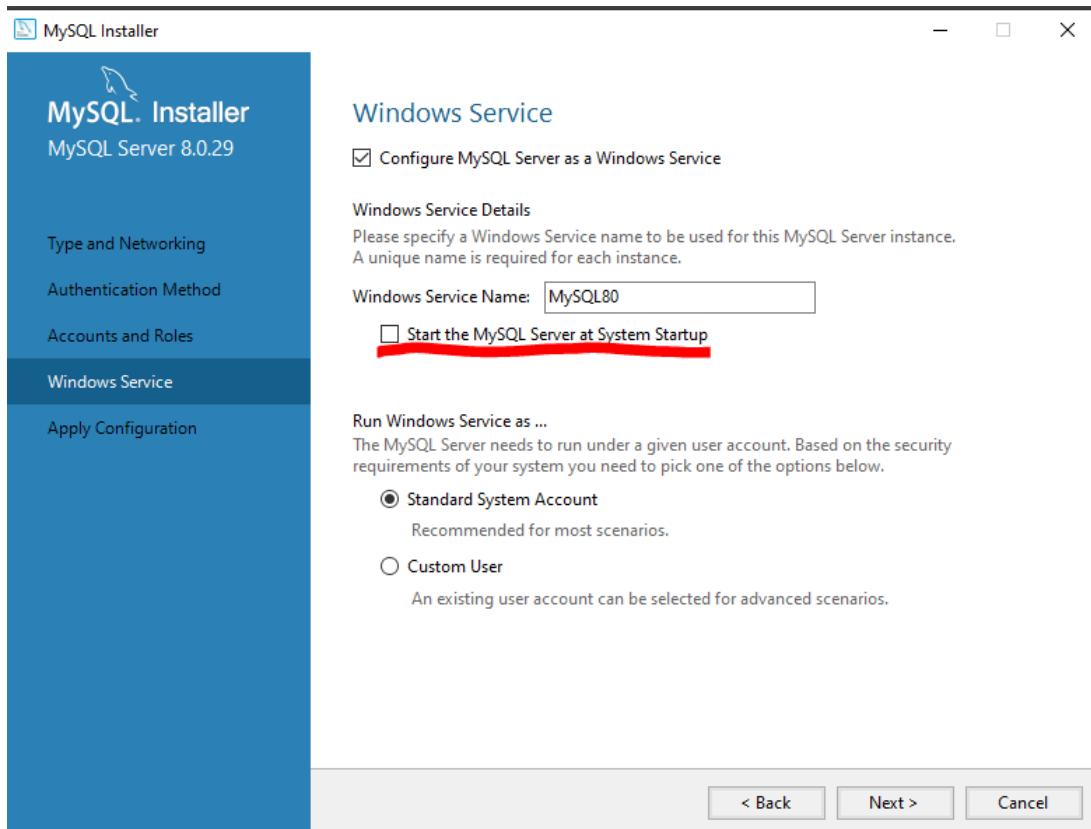
Pasaremos entonces a la configuración. No cambiamos nada:



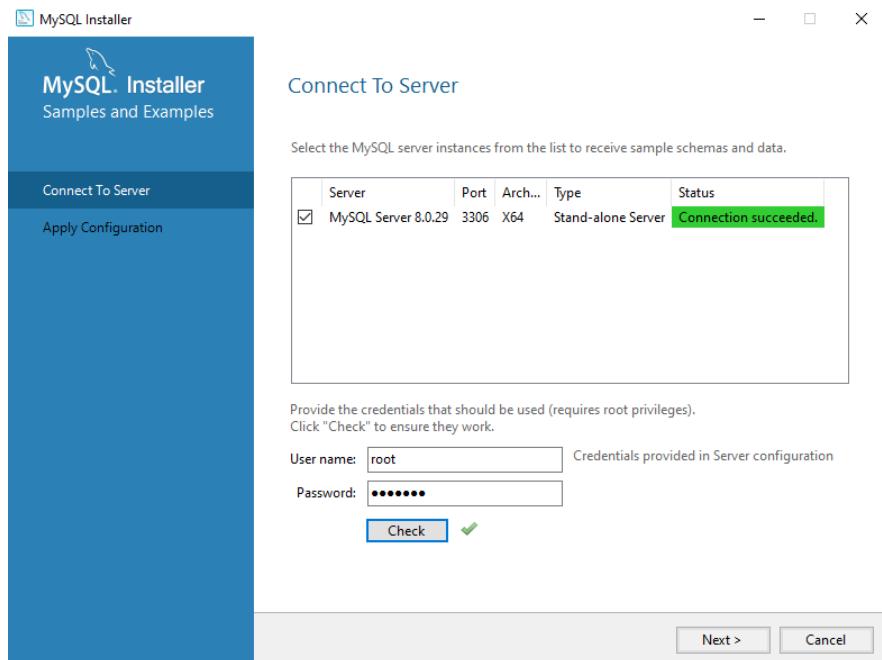
Seguimos en la configuración, asignando contraseña al usuario root, en nuestro caso *abc123*.



Desmarcamos que arranque al iniciar el sistema:



Instalamos los ejemplos, conectándonos como root:

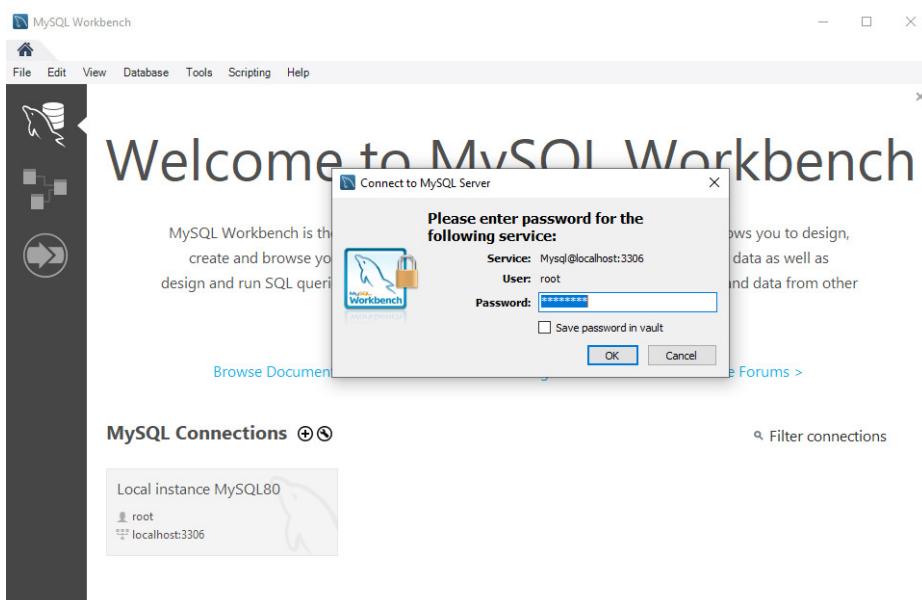


Una vez finalizada la instalación, podemos crear un acceso directo en el escritorio al ejecutable del servidor MySQL para arrancarlo y pararlo y otro acceso directo al ejecutable de MySQL Workbench para las operaciones con las bases de datos. En la instalación por defecto, el primero está en y el segundo en *C:\Program Files\MySQL\MySQL Workbench 8.0*

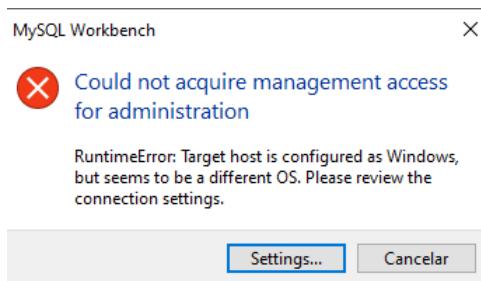
Operaciones habituales

Conexión al servidor

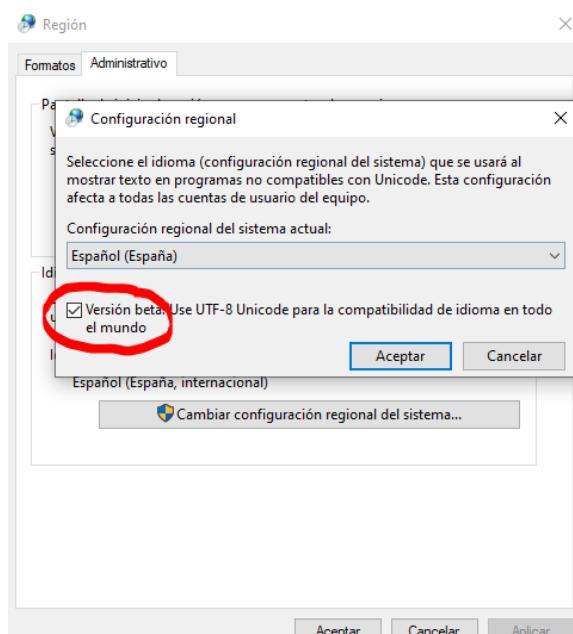
Para arrancar el servidor iremos a los servicios de Windows, buscaremos el servicio MySQL80 (o como le hayamos llamado en la instalación) y lo iniciamos. Ahora usaremos MySQL Workbench para todas nuestras operaciones. Lo primero que debemos hacer es seleccionar nuestro servidor para conectarnos a él (nos pedirá la contraseña de root):



La siguiente operación a realizar es comprobar el estado del servidor: Menú superior *Server* > *Server Status*. Podemos recibir este error:



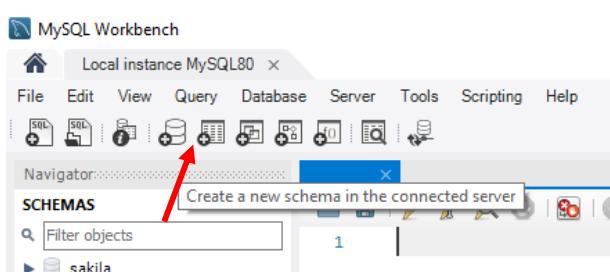
En ese caso, nos dirigimos al Panel de Control de Windows > Región. Pestaña *Administrativo*. Botón *[Cambiar configuración regional del sistema...]* y marcamos el check: *Versión beta. Use UTF-8 Unicode*. Reiniciamos el sistema y reiniciamos el servicio MySQL80.



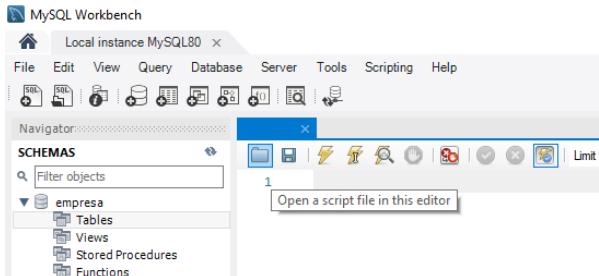
Creación de la Base de Datos

Hibernate necesita trabajar sobre una base de datos creada previamente, aunque no es necesario que contenga ni tablas ni datos ya que es el propio Hibernate puede crearlas en el momento de iniciar la aplicación (este comportamiento lo controlamos mediante el parámetro *spring.jpa.hibernate.ddl-auto* del archivo *application.properties*). Para crearla:

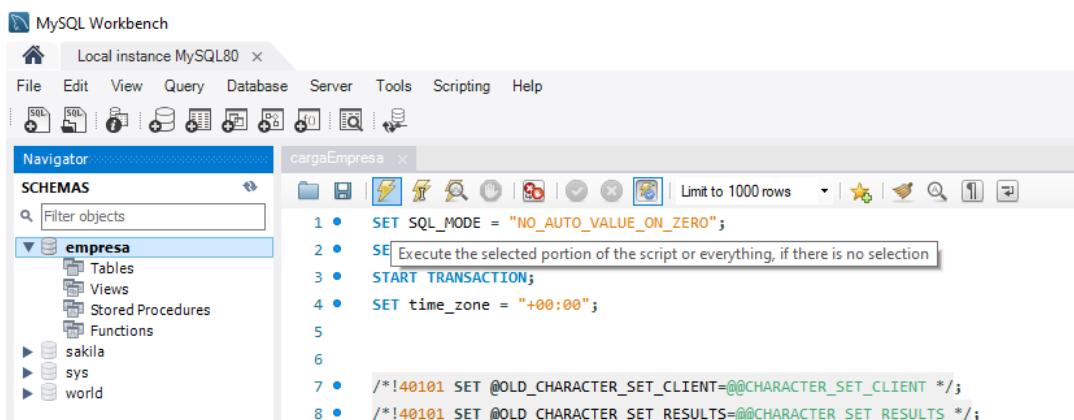
1. Abrir MySQL Workbench y conectarse al servidor.
2. Crear una nueva base de datos (un nuevo esquema) desde el siguiente ícono y asignándole un nombre, por ejemplo: 'empresa' o mediante la sentencia: `create database empresa;`



3. Ahora podríamos crear las tablas y llenar los datos mediante el lenguaje SQL, por ejemplo: "create table nombreTabla...", "insert into nombreTabla values ()", etc. pero sabemos que Hibernate crea las tablas por nosotros. También lo podríamos hacer mediante un archivo que ya tuviese todas esas instrucciones SQL preparadas en forma de script, para que las ejecute. Para ello, en el editor de SQL, seleccionamos el icono de abrir un script y seleccionamos el archivo .sql que contenga las instrucciones (create, insert, etc...)



Haríamos doble clic sobre nuestra base de datos en el panel lateral para seleccionarla y pulsamos en el icono del "rayo" para ejecutarlo.



Exportar / Importar el esquema de la base de datos

En algunas ocasiones nos interesaría modificar el esquema de base de datos creado por Hibernate, por ejemplo, para añadir un índice a una tabla, para añadir una nueva restricción a una columna, etc. Para ello, en vez de construir el esquema desde cero, podemos hacer una primera ejecución de nuestra aplicación para que Hibernate haga el "trabajo sucio" y modificar ese esquema. Los pasos serían:

- Ejecutar la aplicación para que Hibernate genere el esquema de base de datos.
- Exportar el esquema a archivo. Desde MySQL Workbench, menú superior *Server > Data export*. Seleccionamos el esquema y sus tablas y marcamos las opciones *Export to Self-Contained File* y *Include create schema*. Obtenremos un archivo con extensión *.sql*.
- Modificar el archivo anterior con los cambios que deseemos en el esquema.
- Borrar el esquema actual. Desde MySQL Workbench, botón derecho sobre la base de datos > *Drop schema*.
- Ejecutar el archivo con el esquema para que regenere con los nuevos cambios. Desde MySQL Workbench, menú superior *Server > Data import*. Opción: *Import from Self-Contained file* y seleccionamos el archivo anterior.
- Ahora debemos configurar Hibernate para que no regenere el esquema cada vez que se ejecuta. Hemos visto cómo hacerlo para H2 y en el siguiente apartado se recuerda para MySQL.

Parar el servidor

Para finalizar la sesión, de nuevo desde los Servicios de Windows, buscamos el servicio MySQL, botón derecho > Detener.

Configuración del proyecto

Al igual que con H2, la configuración se hace en dos partes: en el pom.xml añadimos la dependencia necesaria y en el fichero *application.properties* establecemos la conexión y configuramos los parámetros de uso.

MySQL en pom.xml

Añadir al fichero pom.xml las dependencias **starter-jpa** y la de la base de datos con la que trabajaremos, en el tema de acceso a datos vimos H2, ahora lo configuraremos para MySQL:

```
<dependency>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-data-jpa</artifactId>
</dependency>
<dependency>
  <groupId>com.mysql</groupId>
  <artifactId>mysql-connector-j</artifactId>
  <version>8.2.0</version>
</dependency>
```

MySQL en application.properties

Debemos indicarle en este archivo la URL de conexión al servidor MySQL y otros parámetros, igual que hicimos con la base de datos H2:

```
spring.datasource.url=jdbc:mysql://localhost:3306/nombreBD
spring.datasource.username=root
spring.datasource.password=abc123.
spring.jpa.hibernate.ddl-auto=create
spring.jpa.properties.hibernate.dialect=org.hibernate.dialect.MySQLDialect
spring.jpa.show-sql=true
```

Como ya vimos, la propiedad *spring.jpa.hibernate.ddl-auto* con el valor *create* hace que recreé las tablas del esquema en cada ejecución. La pasaremos a *validate* o *none* cuando el esquema de base de datos ya no tenga más cambios y queramos conservarlo, incluso con los datos introducidos. Obviamente si realizamos una carga inicial de datos, bien con un archivo *data.sql*, bien con un *CommandLine Runner*, solo debemos hacer esta carga una sola vez y luego eliminar cualquiera de los dos procesos de carga inicial empleados.

Recordemos que la base de datos vacía debe ser creada previamente, por ejemplo, con MySQL Workbench. Hibernate creará las tablas al tener la opción: *spring.jpa.hibernate.ddl-auto=create*.

Testing

Como ya sabemos, el testing consiste en verificar que nuestras aplicaciones funcionan correctamente, esto es, sin errores y respondiendo a las funcionalidades requeridas. Lo primero que debemos hacer es elaborar una estrategia de pruebas, eligiendo los casos a testear.

Dentro del testing, distinguimos pruebas unitarias y pruebas de integración como describiremos más adelante y Spring nos ofrece soporte para implementar ambos tipos de prueba de una forma sencilla, empleando las librerías más populares para esta misión: principalmente JUnit y Mockito.

Sería muy largo de explicar JUnit desde cero, así que puedes ver este video para ponerte al día de lo que es JUnit y luego continuar con este apartado: https://www.youtube.com/watch?v=ZOGz_1XtTKc

Para incluir test en una aplicación Spring Boot es necesaria la dependencia: **starter-test**. Por defecto, esta dependencia incluye el módulo JUnit5 Vintage que da soporte a test JUnit4 por lo que si queremos conservar pruebas de esta versión anterior deberíamos incluir la dependencia *junit-vintage-engine*. También incluye Mockito (librerías para simular la ejecución de métodos, como veremos más adelante), Hamcrest y JAssert. Dentro de la dependencia starter están incluidas otras necesarias como las siguientes: *json-path*, *json-path-assert*, *rest-assured*...

El proyecto tiene una carpeta `/src/test/` con una estructura similar a `/src/main/` donde trabajamos con nuestra aplicación habitualmente. En caso de necesitar un entorno diferente, por ejemplo, usar otra base de datos con datos no reales (e incluso sobre otro gestor de base de datos), podemos incluir un archivo de configuración distinto al de ejecución de la aplicación, en vez de cargar ring Boot carga automáticamente el archivo de configuración `src/test/resources/application.properties`.

Test unitarios

Estos tests van a permitir verificar una unidad o módulo de nuestro código de forma independiente. En general, esa unidad será un método de la aplicación, y probaremos las posibles distintas casuísticas del mismo, es decir, le proporcionaremos distintas entradas para probar que funciona en todas ellas.

Un ejemplo sencillo sería un método que nos devuelve si un número es par o no; deberíamos hacer al menos un test unitario proporcionándole un número par y comprobar que devuelve `true` y otro test proporcionándole uno impar comprobando que devuelve `false`. En resumen, estos test se encargan de probar y verificar métodos de forma aislada del resto de código con el que se relaciona.

Para realizar estas pruebas en Spring, dentro de la carpeta de tests crearemos una estructura de carpetas análoga a la empleada en la aplicación: services, controllers, etc. que contendrán los test unitarios de cada una de las clases que queremos testear. Para cada clase a testear, crearemos una clase con su mismo nombre añadiendo el sufijo Test, por ejemplo: `EmpleadoServiceTest` para testear los métodos de `EmpleadoService`.

Dentro de cada clase de test, precedidos de la anotación `@Test`, añadimos todos los test que queremos hacer sobre los métodos de la misma.

En los tests ejecutaremos el código a probar y contrastaremos el resultado obtenido respecto al esperado mediante una instrucción `assert` como es `assertEquals`, `assertTrue`, etc.

A modo de ejemplo, vamos a crear un servicio con dos métodos sencillos: `sumar()` y `dividir()`, que realizan cálculos sencillos y que lanzan excepciones en caso de que los parámetros no sean correctos.

```

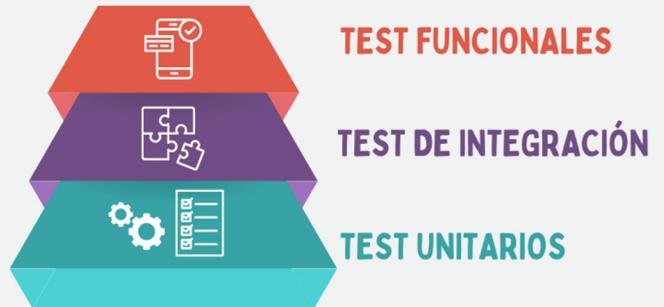
@Service
public class MainService {
    public Integer sumar(Integer op1, Integer op2) {
        if (op1 > 0 && op2 > 0)
            return op1 + op2;
        else throw new IllegalArgumentException("Los dos operandos deben ser positivos");
    }

    public Float dividir(Integer op1, Integer op2) {
        if (op2 != 0)
            return (float) op1 / op2;
        else throw new IllegalArgumentException("No se puede dividir por cero");
    }
}

```

Para probar estos métodos podríamos crear tests que tratasen todas las situaciones posibles, tanto que la operación se ejecuta correctamente como que se produce excepciones. Para este último caso, emplearemos `assertThrows`.

TIPOLOGÍAS DE TESTS



```

@SpringBootTest
@TestInstance(Lifecycle.PER_CLASS)
public class MainServiceTest {
    @Autowired
    private MainService mainService;
    Integer a, b;
    @BeforeEach
    public void init() { a = 3; b = 2; }
    @Test
    public void sumarTest_ok() {
        assertEquals(5, mainService.sumar(a, b));
    }
    @Test
    public void sumarTest_except() {
        a = -1;
        assertThrows(IllegalArgumentException.class, () -> { mainService.sumar(a, b); });
    }
    @Test
    public void dividirTest() {
        assertEquals(1.5f, mainService.dividir(a, b));
    }
}

```

Vemos como tenemos la anotación `@Test` para cada test que realizamos, en este caso son tres tests y hemos incluido un método con la anotación `@BeforeEach` que se ejecutará antes de cada test. Podríamos incluir otras anotaciones JUnit: `@BeforeAll`, `@AfterAll`, que es ejecutarían, como indica su nombre antes del conjunto de tests o a finaliza el conjunto de tests.

Hay que destacar dos consideraciones sobre las pruebas a nivel operativo:

- **Las pruebas deben ser independientes** unas de otras, es decir, que el resultado o cálculos de un test no deben ser empleados en otro test. De hecho, si no incluimos la anotación `@Order(1)`, `@Order(2)` etc... a cada test, no se garantiza que se ejecuten en un orden determinado.
- **Las pruebas deben ser idempotentes**, es decir, que se pueden ejecutar una o varias veces y siempre obtendremos el mismo resultado. Para ello es necesario que, si las pruebas han modificado algún recurso, éste sea restaurado a su situación inicial al finalizar la prueba. Por ejemplo: si la prueba crea una instancia de cliente en el repositorio de clientes, al terminar la prueba la borraremos.

Ejecución de los tests:

Si trabajamos con Visual Studio Code, se pueden ejecutar los tests clicando en el ícono (círculo verde) que aparece a la izquierda cada test. Si clicamos en ese mismo ícono, en el nombre de la clase de test ejecutaría todas las pruebas definidas para esa clase.

```

38  @SpringBootTest
39  @TestInstance(Lifecycle.PER_CLASS)
40  @AutoConfigureMockMvc
41  public class CuentaControllerTest {
42      @Autowired
43      private MockMvc mockMvc;

```

En proyectos Maven, una forma muy cómoda de ejecutar todos los test del proyecto es mediante el comando: `mvn test`

Mocking de capas

Como nuestras aplicaciones están organizadas en capas: controlador, servicio, repositorio, etc. cuando realizamos un test unitario de un método de una capa, debemos “simular” (en inglés: *mock*) las capas dependientes, para que nuestros test sean realmente “unitarios” y no de integración de capas. Para ello, partiendo de una clase de test como la que acabamos de ver como ejemplo, haríamos los siguientes cambios:

- Incluir la clase a testear precedida de la anotación `@InjectMocks` para indicarle alguna o todas de las clases dependientes que empleará serán falseadas. Debe ser siempre una clase, no una interfaz, así que en aquellos casos que tenemos ambas (por ejemplo: la interfaz `EmpleadoService` y la clase que la implementa `EmpleadoServiceImpl`, debemos anotar la clase).
- Incluir las clases dependientes de la clase a testear que queremos que no ejecuten su versión real si no la falseada. Irán precedidas de la anotación de Mockito: `@Mock` en el caso de los repositorios y de la anotación de `SpringBoot @MockBean` en el caso de los servicios.

Lo vemos más claro en el siguiente ejemplo, en el que vamos a testear una clase de servicio, que generalmente tiene inyectado un repositorio. Cuando queremos probar solo el servicio no queremos que se haga una llamada a los métodos reales del repositorio, ya que no queremos que errores en el repositorio puedan afectar al resultado de la prueba del servicio.

Testing del Servicio

El siguiente ejemplo se basa en el proyecto de CRUD de Empleado visto durante este manual y testearía el método `obtenerTodos()` del servicio, que a su vez llama al `findAll()` del repositorio de empleados.

```
import static org.junit.jupiter.api.Assertions.assertEquals;
import static org.mockito.Mockito.*; //importa: times, verify, when...

//resto de imports (no static)

@SpringBootTest
@ExtendWith(MockitoExtension.class)
public class EmpleadoServiceTest {
    ArrayList<Empleado> mockList; //variable que usaremos en los tests
    @InjectMocks
    EmpleadoServiceImpl empleadoService;

    @Mock
    EmpleadoRepository empleadoRepository;

    @BeforeAll //inicialización previa al conjunto de tests
    public void init() {
        mockList = new ArrayList<>();
        mockList.add(new Empleado(1L, "test1", "test1@gmail.com", 1000d, true, Genero.MASCULINO));
        mockList.add(new Empleado(2L, "test2", "test2@gmail.com", 2000d, true, Genero.FEMENINO));
    }

    @Test
    public void obtenerTodosTest() {
        when(empleadoRepository.findAll()).thenReturn(mockList);
        List<Empleado> empList = empleadoService.obtenerTodos();
        assertEquals(2, empList.size());
        verify(empleadoRepository, times(1)).findAll();
    }
}
```

Vamos a analizarlo paso a paso: primero se referencia con `@InjectMocks` la clase a testear, en este caso `EmpleadoService`, luego se crea una instancia de la dependencia `empleadoRepository` que “falsearemos” mediante `@Mock` para que, cuando el método de servicio llame a un método del repositorio, no sea una real, si no que devuelva los datos falsos incluidos en el propio test.

A continuación, en la sección `@BeforeAll`, que se ejecuta al principio de todo el conjunto de pruebas, inicializa un arraylist de empleados que usaremos en los tests siguientes para simular la llamada al método `findAll` del `JpaRepository` de empleados.

El siguiente paso es el test en sí, anotado con `@Test`. La cláusula `when` hace que cuando se invoque al `findAll()` del repositorio devuelva la lista creada en vez de los datos reales del repositorio.

Finalmente, invocamos al método del servicio que estamos testeando, que llamará al método `findAll()` del repositorio (pero el falso!), y ya podemos hacer los `asserts` de validación que deseemos. En este caso se verifica mediante `assertEquals` que se han devuelto 2 empleados.

```
assertEquals(2, empList.size());
```

Otra validación, mediante `verify`, consiste en validar el número de veces que se ha llamado al método del repositorio, en este caso debe ser una sola vez.

```
verify(empleadoRepository, times(1)).findAll();
```

Podemos añadir nuevos tests a esta clase, para el resto de métodos del servicio, como por ejemplo: `obtenerPorId()`, `añadir()` o `borrar()`.

```
@Test
public void obtenerPorIdTest() {
    when(empleadoRepository.findById(1L)).thenReturn(Optional.of(mockList.get(0)));
    Empleado empleado = empleadoService.obtenerPorId(1L);
    assertEquals("test1", empleado.getNombre());
    assertEquals("test1@gmail.com", empleado.getEmail());
    assertEquals(1000, empleado.getSalario());
}

@Test
public void añadirTest() {
    Empleado empleadoNew = new Empleado(3L, "test3", "test3@gmail.com", 1000d, true,
        Genero.MASCULINO);
    when(empleadoRepository.save(empleadoNew)).thenReturn(empleadoNew);
    Empleado insertado = empleadoService.añadir(empleadoNew);
    assertEquals("test3", insertado.getNombre());
    verify(empleadoRepository, times(1)).save(empleado1);
}

@Test
public void borrarTest_ok() {
    when(empleadoRepository.findById(1L))
        .thenReturn(Optional.of(mockList.get(0)));
    empleadoService.borrar(1L);
    verify(empleadoRepository, times(1)).findById(1L);
    verify(empleadoRepository, times(1)).delete(mockList.get(0));
}

@Test
public void borrarTest_notFound() {
    when(empleadoRepository.findById(999L))
        .thenReturn(Optional.ofNullable(null));
    empleadoService.borrar(999L);
    verify(empleadoRepository, times(1)).findById(999L);
    verify(empleadoRepository, times(0)).delete(null);
}
```

Hay que resaltar que hay dos tests para el borrado, uno en el que sí se borra (se encontró el empleado con el `findById` previo que se hace en el método de servicio de borrado) y otro que no borra porque no lo encuentra (`findById` devuelve un `Optional` de nulo).

Si hubiera más reglas de negocio en el servicio, deberíamos implementar más tests. Imaginemos que no se pudiese dar de alta un empleado con un salario de más de 10000 euros y que se produjese una excepción. Deberíamos probar el caso también con `assertThrow`.

Testing del Controlador API REST

La filosofía de un test unitario de controlador es similar a la de un test de servicio, pero debemos tener en cuenta las siguientes consideraciones:

- De forma análoga a lo que ocurre con los test de servicio, en un test de controlador debemos injectar la clase a testear con la anotación `@InjectMocks` y las clases dependientes falseadas con `@MockBean` (en general, será la clase de servicio la falseada).
- Necesitaremos una instancia de la clase `MockMvc` que se encargará de hacer las llamadas a los métodos de controlador, es decir, que simulará las llamadas a la API desde el cliente. Podremos hacer llamadas de tipo get, post, put, etc...
- Disponemos opcionalmente de la anotación `@WebMvcTest` para que, en caso de que nuestra aplicación conste de varios controladores, solo cargue el controlador especificado y sus dependencias sin cargar toda la aplicación. Ejemplo: `@WebMvcTest(HomeController.class)`.

Vamos a ver un ejemplo sencillo, de un `RestController` que recibe en el path de la URL dos números y llama a un método que devuelve su suma:

```
@RestController
public class EjemploController {
    @Autowired
    private EjemploService ejemploService;

    @GetMapping("/suma/{id1}/{id2}")
    public ResponseEntity<?> getSuma(@PathVariable Integer id1, @PathVariable Integer id2) {
        Integer suma = ejemploService.sumar(id1, id2);
        return ResponseEntity.ok(new Respuesta(id1, id2, suma)); // cod 200
    }
}
```

Siendo la clase `Respuesta`:

```
@AllArgsConstructor
@Data
public class Respuesta {
    private Integer sumando1;
    private Integer sumando2;
    private Integer resultado;
}
```

Un posible test del método del controlador podría ser el siguiente:

```
import static org.hamcrest.core.Is.is;
import static org.mockito.Mockito.when;
import static org.springframework.test.web.servlet.request.MockMvcRequestBuilders.*;
import static org.springframework.test.web.servlet.result.MockMvcResultMatchers.*;
//resto de imports (no static)
@SpringBootTest
@AutoConfigureJsonTesters
@AutoConfigureMockMvc
@TestInstance(Lifecycle.PER_CLASS)
public class EjemploControllerTest {
    @InjectMocks
    EjemploController ejemploController;

    @MockBean
    EjemploService ejemploService;

    @Autowired
    MockMvc mockMvc;
```

```

    @Test
    public void getSumaTest() throws Exception {
        when(ejemploService.sumar(2, 3)).thenReturn(5);

        mockMvc.perform(get("/suma/2/3")
            .contentType(MediaType.APPLICATION_JSON))
            .andExpect(status().isOk())
            .andExpect(jsonPath("$.resultado", is(5)));
    }
}

```

Vemos como inyecta el propio controlador y el servicio falsoado que utiliza el primero. Y por otra parte inyecta una instancia de `MockMvc` para simular las llamadas al controlador. El método `perform` acompañado del verbo HTTP y la URL, es el que realmente hace la llamada al método del controlador, en el ejemplo, hace una petición GET a la ruta "/suma/2/3".

Mediante las cláusulas `andExpect` indicamos tanto el estado de la respuesta esperado (Ok-200, Created-201, etc). como datos a validar en el cuerpo de la respuesta. `jsonPath $` representa el objeto devuelto, en este caso de tipo "*Respuesta*". Si la respuesta recibida fuese una colección, por ejemplo, de 3 elementos, podríamos incluir validación siguiente:

```

        .andExpect(jsonPath("$.size()", hasSize(3))
        .andExpect(jsonPath("$.[0].email", is("test1@gmail.com")))
        .andExpect(jsonPath("$.[1].email", is("test2@gmail.com")))
        .andExpect(jsonPath("$.[2].email", is("test3@gmail.com")));

```

Veamos ahora como sería un ejemplo de validación de un `RestController` de un CRUD completo. Seguimos con el ejemplo de la entidad `Empleado` seguido a lo largo de este manual.

```

import static org.hamcrest.Matchers.*;      //hasSize, instanceof, put...
import static org.hamcrest.core.Is.is;
import static org.mockito.Mockito.when;
import static org.mockito.Mockito.doNothing;
import static org.springframework.test.web.servlet.request.MockMvcRequestBuilders.*; //get, post, put...
import static org.springframework.test.web.servlet.result.MockMvcResultMatchers.*; //jsonPath, status...

//resto de imports (no static)

@SpringBootTest
@AutoConfigureJsonTesters
@AutoConfigureMockMvc
@TestInstance(Lifecycle.PER_CLASS)
public class EmpleadoControllerTest {
    List<Empleado> empleados;      //variables que emplearemos en varios tests
    Empleado empleadoSinId;

    @InjectMocks
    EmpleadoController empleadoController;

    @MockBean
    EmpleadoService empleadoService;

    @Autowired
    MockMvc mockMvc;

    @BeforeAll
    void initTest() {
        empleados = new ArrayList<>();
        empleados.add(new Empleado(1L, "pepe", "pepe@gmail.com", 800d, true, Genero.MASCULINO));
        empleados.add(new Empleado(2L, "ana", "ana@gmail.com", 900d, true, Genero.FEMENINO));
        empleados.add(new Empleado(3L, "luis", "luis@gmail.com", 2000d, true, Genero.FEMENINO));
        empleadoSinId = new Empleado(0L, "pepe", "pepe@gmail.com", 800d, true, Genero.MASCULINO);
    }
}

```

```

@Text
public void getAllEmpleadoTest() throws Exception {
    when(empleadoService.obtenerTodos()).thenReturn(empleados);
    mockMvc.perform(get("/empleado"))
        .contentType(MediaType.APPLICATION_JSON)
        .andExpect(status().isOk())
        .andExpect(jsonPath("$.size()", hasSize(3)))
        .andExpect(jsonPath("$.id", is(1)))
        .andExpect(jsonPath("$.nombre", is("pepe")))
        .andExpect(jsonPath("$.email", is("pepe@gmail.com")))
        .andExpect(jsonPath("$.salario", is(900d)));
}

@Text
public void getOneEmpleadoTest() throws Exception {
    when(empleadoService.obtenerPorId(1)).thenReturn(empleados.get(0));
    mockMvc.perform(get("/empleado/1"))
        .contentType(MediaType.APPLICATION_JSON)
        .andExpect(status().isOk())
        .andExpect(jsonPath("$.nombre", is("pepe")))
        .andExpect(jsonPath("$.salario", is(800d)));
}

@Text //test del alta de empleado: método POST
public void addEmpleadoTest() throws Exception {
    when(empleadoService.agregar(empleadoSinId)).thenReturn(empleados.get(0));
    mockMvc.perform(post("/empleado/"))
        .contentType(MediaType.APPLICATION_JSON)
        .content(new ObjectMapper().writeValueAsString(empleadoSinId))
        .andExpect(status().isCreated())
        .andExpect(jsonPath("$.id", is(1)))
        .andExpect(jsonPath("$.nombre", is("pepe")))
        .andExpect(jsonPath("$.salario", is(800d)));
}

@Text
public void deleteOneEmpleadoTest() throws Exception {
    //when (empleadoService.obtenerPorId(1)).thenReturn(empleados.get(0));
    doNothing().when(empleadoService).delete(1L);
    mockMvc.perform(delete("/empleado/1"))
        .contentType(MediaType.APPLICATION_JSON)
        .andExpect(status().isNoContent());
}
}

```

Testing del Controlador MVC / Thymeleaf

El proceso para este tipo de tests es similar al anterior, con la diferencia de que, en vez de recibir un archivo JSON como respuesta, será una vista con un modelo asociado que contiene los datos.

Para el envío de datos desde el cliente, con POST por ejemplo, no será mediante archivos JSON, al ser mediante un formulario lo haremos mediante *params*, como veremos en el ejemplo.

Podremos validar que la vista devuelta es la correcta y que los valores de los atributos del *model* también son los correctos, para ello emplearemos el objeto **MockMvcResultMatchers**. Esta clase dispone de métodos como **view()**, **model()** para acceder a los elementos recibidos. Lo vemos en el siguiente ejemplo:

```

import static org.hamcrest.Matchers.*;           //importa: allOf, equalTo, hasItem, hasProperty, hasSize, instanceOf...
import static org.springframework.test.web.servlet.request.MockMvcRequestBuilders.*;        //importa: get, post...
import static org.springframework.test.web.servlet.result.MockMvcResultMatchers.*;           //importa model, view, status, redirectedUrl...
//resto imports...

```

```

@SpringBootTest
@则TestInstance(Lifecycle.PER_CLASS)
@AutoConfigureMockMvc
public class MainControllerTest {
    @Autowired
    private MockMvc mockMvc;

    @InjectMocks
    MainController mainController; //clase a testar con servicio dependiente mockeado
    @MockBean
    MainService mainService; // clase/interfaz falseada con cláusulas when

    @Test
    void addElementTest() throws Exception {
        when(mainService.añadir(new Persona("Pepe",20)).thenReturn(true);

        mockMvc.perform(post("/addUrl/new/submit")
            .contentType(MediaType.APPLICATION_FORM_URLENCODED)
            .param("nombre", "Pepe")
            .param("edad", "20"))
            .andExpect(status().is3xxRedirection())
            .andExpect(redirectedUrl("/home/")); // también redirectedUrlPattern("/myURL/*"))
    }

    @Test
    void listElementsTest() throws Exception {
        ArrayList <Persona> lista = new ArrayList<>();
        lista.add (new Persona("Pepe",20));
        when(mainService.obtenerTodos()).thenReturn(lista);
        mockMvc.perform(get("/home/"))
            .andExpect(status().isOk())
            .andExpect(view().name("homeView"))
            .andExpect(model().attributeExists("listaElem"))
            .andExpect(model().attribute("listaElem", instanceof(ArrayList.class)))
            .andExpect(model().attribute("listaElem", hasSize(1)))
            .andExpect(model().attribute("listaElem", hasItem(allOf(
                hasProperty("nombre", equalTo("Pepe")),
                hasProperty("saldo", equalTo(20))))));
    }
}

```

Podemos ver que sobre el *model*/de la vista recibido podemos hacer distintas validaciones:

- **model().attributeExists()**: comprueba si el *model* contiene el atributo.
- **model().attribute("nombreAtributo", instaceOf(Clase.class))** : comprueba que el atributo del modelo de un tipo determinado: ArrayList.class, Empleado.class, etc.
- **model().attribute("nombreAtributo", "valor")**: equivalente a un *assertEquals*, es decir, comprueba si el atributo del *model* tiene el valor indicado.

En el caso de que el elemento del *model*/sea una colección, disponemos de otras validaciones como:

- **model().attribute("lista", hasSize(X))**: comprueba si la lista tiene X elementos.
- **model().attribute("lista", hasItem(E))**: comprueba si la lista contiene un elemento E.
- **model().attribute("lista", hasItem(allOf(hasProperty...)))** : comprueba si la lista contiene un elemento que tenga todas las propiedades indicadas en los *hasProperty*.

Cuando queremos hacer un post (simulando el envío de un formulario), si este incluye un archivo, en el método *perform* de la clase *MockMvc* se emplea **multipart** en vez de *post* y no se informa el *contentType*. El siguiente ejemplo hace el test de un formulario que envía un nombre y una imagen.

```

byte[] imageBytes = Files.readAllBytes(Paths.get("path/imagen.png"));
MockMultipartFile myimg= new MockMultipartFile(
    "file", // El nombre del campo del formulario
    "path/imagen.png", // Nombre del archivo
    "image/png", // Tipo de contenido
    imageBytes // Contenido del archivo
);
mockMvc.perform(multipart("/form/submit")
    .file(myimg)
    .param("nombre", "Ana"))
    .andExpect(status().isOk());

```

Para testear mappings de controlador que necesitan estar autenticados en la aplicación y tener los permisos adecuados, se debe anotar el test con `@WithMockUser`. Así, la ejecución se hará bajo un usuario ficticio con nombre '*user*' y rol '*USER*'. Si se necesita otro tipo de rol se puede especificar, como se puede ver en el siguiente ejemplo. Asimismo, los métodos de envío de formulario deben llevar: `.with(csrf())`. Ejemplo:

```

@Test
@WithMockUser(roles = "ADMIN")
void myTest() throws Exception {
    mockMvc.perform(post("/adminform/submit")
        .contentType(MediaType.APPLICATION_FORM_URLENCODED)
        .with(csrf())
        .param("formfield", "value"))
        .andExpect(status().isOk());
}

```

Tests de integración y funcionales

Los test de integración son aquellos que prueban varias capas a la vez. Su objetivo es ver y analizar los posibles defectos y errores en la interacción y comunicación entre las diferentes capas. Se diferencian de los test unitarios en que la petición es más amplia, no falseando todas las capas. Por ejemplo, se podría hacer un test de integración entre un método de controlador, sin falsear la lógica de negocio, pero falseando la capa de repositorio.

Los test funcionales, también llamados **test de aceptación**, son aquellos que prueban todas capas como un todo, es decir, que la funcionalidad del caso de uso del cliente funciona correctamente en todas sus partes (interacción con el usuario, lógica de negocio, acceso a datos, etc.). No hay ninguna capa de la aplicación en su versión "*mock*", así pues, serían similares a un test de controlador (con una instancia de `MockMvc`) pero incluyendo el resto de capas reales, sin `@InjectMocks`, `@MockBean`...

Son las pruebas más importantes, ya que prueban la aplicación en su conjunto. Si no disponemos de tiempo o recursos para la fase de testing de nuestros proyectos, deberíamos al menos implementar los test funcionales de las partes más críticas de la aplicación; por ejemplo, en una tienda virtual, un test que crease un carrito de la compra y añadiese algún producto.

Más información sobre testing:

<https://refactorizando.com/ejemplos-testing-spring-boot/>
<https://www.baeldung.com/spring-boot-testing>
<https://www.javaguides.net/2022/03/spring-boot-unit-testing-crud-rest-api-with-junit-and-mockito.html?m=1>

Perfiles de configuración

Spring permite configurar de distinta forma los distintos entornos en lo que se puede ejecutar la aplicación, a saber: desarrollo, integración o testing, producción, etc. Un ejemplo típico suele ser el gestor de base de datos empleado, podríamos usar H2 en desarrollo y testing y MySQL en producción. También podemos hacer que ciertos componentes (beans) estén disponibles en uno y otro entorno.

La forma más sencilla de configurar distintos entornos, cada uno con sus características, es crear distintos archivos de propiedades del proyecto, uno para cada perfil. Así, dejaríamos en `application.properties` las propiedades comunes a todos los perfiles, y luego crearíamos archivos con nombre `application-perfil.properties`, por ejemplo:

```
application-dev.properties  
application-prod.properties
```

Para establecer cuál de los perfiles está activo en cada momento, añadiremos al archivo `application.properties` general la propiedad `spring.profiles.active` con el perfil activo, por ejemplo:

```
spring.profiles.active=dev
```

Existen otras formas de definir el perfil activo, mediante una variable de entorno del sistema, mediante parámetros de la JVM o mediante el `pom.xml`.

Adicionalmente, podríamos crear componentes que solo estuviesen disponibles en un determinado perfil. Para ello, añadiríamos la anotación `@Profile` con perfil en el que el componente está disponible. Por componentes entendemos todos los `@Bean`, `@Configuration`, `@Component` y sus derivados (`@Controller`, `@Service`, etc.), etc.

```
@Configuration  
@Profile("prod")  
public class ProductionConfiguration { . . . }
```

Otro ejemplo típico puede ser un `CommandLineRunner` que haga una carga inicial, pero solo en test:

```
@Bean  
@Profile("test")  
public CommandLineRunner initData(ProductRepository pr) {  
    return args -> {  
        pr.add(new Product("Prod 1", 100.0f));  
        pr.add(new Product("Prod 2", 200.0f));  
    };  
}
```

Logging

Por `"/logging"` o "registro" nos referimos a la publicación de información sobre los distintos eventos que ocurren durante la ejecución nuestra aplicación. Esta información sirve para identificar qué está pasando en el sistema, incluyendo errores, problemas o avisos menores, y en qué momento han sucedido. Por defecto, los logs se muestran por la terminal, pero podemos configurarlos para que sean guardados en ficheros. En entorno de producción, el logging es vital para encontrar problemas, monitorizar el sistema y comprobar su salud. En entorno de desarrollo, nos ayuda a hacer a comprender el flujo de la aplicación y ayuda al "debug" o detección de errores.

En Spring podemos usar distintas librerías de `logging` (Logback, Log4j2, etc.) siendo Logback la configurada por defecto. Además, disponemos SLF4J (*Simple Logging Facade for Java*) que como su nombre indica es una fachada, actúa como una capa de abstracción que permite usar las mismas llamadas independientemente del sistema de log que tengamos instalado. Usando SLF4J el cambio de sistema de log será transparente para la aplicación.

Niveles de logging

Los mensajes que enviamos a nuestro log se pueden dividir en distintas categorías dependiendo de su importancia. Los niveles estándar son:

- **ERROR:** error que puede hacer que la aplicación deje de funcionar correctamente.
- **WARN:** aviso de un potencial problema que puede llegar a generar problemas mayores y que debemos atender.

- INFO: proporciona información general sobre las tareas que está realizando la aplicación, confirmando que está funcionando como se esperaba.
- DEBUG: ofrece información a los desarrolladores para resolver errores de ejecución. No deberíamos mostrar estos mensajes en entorno de producción.
- TRACE: Similar a DEBUG pero ofreciendo más detalle.

En la configuración fijaremos el nivel de logging que queremos mostrar, incluyendo los niveles superiores al fijado. Es decir, si lo fijamos a *WARN*, veremos también *ERROR* pero no los restantes.

Uso del logger

El sistema de logging ya viene preconfigurado, por lo que, para usarlo, solo debemos crear una variable que gestione el log y luego, cuando queramos añadir un mensaje al log, emplear uno de los métodos que se corresponden a los niveles de log.

Una forma de crear la variable podría ser así:

```
Logger log = LoggerFactory.getLogger(LoggingController.class);
```

Pero podemos simplificar la tarea usando Lombok, mediante la anotación `@Slf4j` para que realice la misma tarea. En este caso la variable será estática y se llamará 'log'.

```
@Slf4j
@Controller
public class MiControlador {
    public String showHome(){
        log.info("Un usuario acaba de acceder a la aplicación");
        return "homeView";
    }
}
```

Los métodos disponibles son:

```
log.trace("mensaje");
log.debug("mensaje");
log.info("mensaje");
log.warn("mensaje");
log.error("mensaje");
```

La siguiente imagen muestra el log de una aplicación en la que sus tres últimas líneas se corresponden a llamadas: `log.error()`, `log.info()` y `log.warn()` situadas en distintos puntos de la aplicación.

```
16:52:09.088+01:00 INFO 6232 --- [ restartedMain] o.s.b.w.embedded.tomcat.TomcatWebServer : Tomcat initialized with port(s): 9000 (http)
16:52:09.134+01:00 INFO 6232 --- [ restartedMain] o.apache.catalina.core.StandardService : Starting service [Tomcat]
16:52:09.138+01:00 INFO 6232 --- [ restartedMain] o.apache.catalina.core.StandardEngine : Starting Servlet engine: [Apache Tomcat/10.1.15]
16:52:09.426+01:00 INFO 6232 --- [ restartedMain] o.a.c.c.C.[Tomcat].[localhost].[] : Initializing Spring embedded WebApplicationContext
16:52:09.470+01:00 INFO 6232 --- [ restartedMain] w.s.c.ServletWebServerApplicationContext : Root WebApplicationContext: initialization completed in 4 ms
16:52:11.757+01:00 INFO 6232 --- [ restartedMain] o.s.b.d.a.OptionalLiveReloadServer : LiveReload server is running on port 35729
16:52:11.876+01:00 INFO 6232 --- [ restartedMain] o.s.b.w.embedded.tomcat.TomcatWebServer : Tomcat started on port(s): 9000 (http) with context path /
16:52:11.912+01:00 INFO 6232 --- [ restartedMain] com.example.empleado.Main : Started Main in 7.592 seconds (process running for 9.185 ms)
16:52:11.927+01:00 INFO 6232 --- [ restartedMain] com.example.empleado.Main : Carga inicial de empleados correcta
16:52:29.720+01:00 INFO 6232 --- [nio-9000-exec-1] o.a.c.c.C.[Tomcat].[localhost].[] : Initializing Spring DispatcherServlet 'dispatcherServlet'
16:52:29.721+01:00 INFO 6232 --- [nio-9000-exec-1] o.s.web.servlet.DispatcherServlet : Initializing Servlet 'dispatcherServlet'
16:52:29.726+01:00 INFO 6232 --- [nio-9000-exec-1] o.s.web.servlet.DispatcherServlet : Completed initialization in 4 ms
16:52:29.846+01:00 ERROR 6232 --- [nio-9000-exec-1] c.e.e.controllers.EmpleadoController : Usuario quiere borrar empleado inexistente
16:52:29.969+01:00 INFO 6232 --- [nio-9000-exec-2] c.e.e.controllers.EmpleadoController : Un usuario acaba de acceder a la aplicación
16:53:40.717+01:00 WARN 6232 --- [nio-9000-exec-6] c.e.e.controllers.EmpleadoController : Usuario envía formulario con errores
```

El nivel de logging por defecto es `INFO`, por lo que los mensajes `TRACE` y `DEBUG` no serán visibles, aunque veremos en el apartado de configuración cómo cambiar este comportamiento.

Un uso típico de los logs es en la gestión de excepciones (en lugar de usar el método `printStackTrace()`):

```

try {
    // código que puede Lanzar una excepción
} catch (Exception ex) {
    log.error("An error occurred while processing", ex);
}

```

Si queremos que los mensajes tengan un contenido dinámico, no es necesario concatenar texto y variables. SLF4J nos ofrece parametrización:

```

String orderId = "012345";
log.info("Processing order with ID: {}", orderId);

```

Configuración

Aunque el sistema de logging funciona perfectamente sin parametrización alguna, podemos configurar el comportamiento del sistema de distintas formas, siendo el *application.properties* un lugar muy apropiado para ello. Estos son los principales parámetros que podemos configurar en ese archivo.

Nivel global de logging: añadimos la propiedad *logging.level.root*:

```
logging.level.root=WARN
```

y también podríamos distinguir distintos niveles para distintas clases o paquetes:

```

logging.level.com.myapp.mypackage=TRACE
logging.level.com.myappmyservice=INFO

```

A la hora de ejecutarlo, podemos cambiar también el comportamiento por defecto, con parámetros:

```
java -jar myApp-0.0.1-SNAPSHOT.jar -trace
```

o bien:

```

mvn spring-boot:run
-Dspring-boot.run.arguments=--logging.level.org.springframework=WARN

```

Guardar el log en fichero: Si queremos que los logs se almacenen en fichero, por ejemplo, llamado *myapp.log* no solo se muestren por consola, basta con añadir al *application.properties* la línea:

```
logging.file.name=myapp.log
```

Disponemos de parámetros adicionales para gestionar el tamaño del archivo de log y rotar su contenido.

Mostrar colores en los mensajes: *spring.output.ansi.enabled=always*

Configuración del formato del log: Usando *logging.pattern.console* y *logging.pattern.file*, puedes definir el formato de salida de cada línea de log, tanto para consola como para fichero, respectivamente. El siguiente ejemplo limitaría el tamaño del nombre del log a 36 caracteres:

```
logging.pattern.console=%d{yyyy-MM-dd HH:mm:ss} - %logger{36} - %msg%n
```

Configuración más detallada

Por último, podríamos hacer una configuración mucho más detallada en archivos específicos de cada sistema de log, por ejemplo, para LogBack mantendríamos en la carpeta */resources* un fichero llamado *logback-spring.xml* o *logback.groovy*.

Para usar el otro sistema de log comentado, Log4j2 debemos eliminar la dependencia de Logback en el *pom.xml* y añadir la de Log4j2.

```

<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-web</artifactId>
    <exclusions>
        <exclusion>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-starter-logging</artifactId>
        </exclusion>
    </exclusions>
</dependency>
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-log4j2</artifactId>
</dependency>

```

Y haríamos la configuración en un archivo llamado *log4j2-spring.xml*

Buenas prácticas

- Registra mensajes significativos:** Asegúrate de que cada mensaje de registro proporciona un contexto y es lo suficientemente claro como para que alguien no familiarizado con el código pueda entenderlo. Deben evitarse mensajes ambiguos como "Se ha producido un error".
- Utiliza niveles de registro adecuados:** Un uso incorrecto de los niveles de registro puede hacer que se pierda información crítica o que se llenen los registros. Utiliza el nivel adecuado: ERROR, WARN,... En un entorno de producción no deberían mostrarse mensajes de DEBUG ni TRACE.
- Evita registrar información sensible:** Nunca registres información sensible como contraseñas, números de tarjetas de crédito o información personal identificable. Esta es una buena práctica de seguridad y, en muchas jurisdicciones, un requisito legal.
- Utiliza mensajes de log parametrizados:** En lugar de la concatenación de cadenas, utiliza el registro parametrizado que proporciona SLF4J. Este enfoque es eficiente y puede evitar la creación innecesaria de cadenas.
- Maneja las excepciones adecuadamente:** Cuando registres excepciones, es crucial que se registre todo el seguimiento de la pila para diagnosticar la causa raíz.
- No dependas únicamente de los logs para la supervisión:** Aunque los logs son muy valiosos para el diagnóstico, no deben ser la única herramienta de supervisión. Las métricas, alertas y otras herramientas de supervisión deben utilizarse junto con los logs.
- Rotación y archivo de logs:** Asegúrate de están configurados para rotar los logs, evitando que un único archivo se haga demasiado grande y garantizando que los archivos de log más antiguos se archiven para posibles análisis futuros.
- Evita el log dentro de bucles:** El registro dentro de bucles puede ralentizar significativamente una aplicación y generar archivos de log enormes.
- Mantén la coherencia:** Mantén la coherencia en el formato de los patrones de los mensajes de log para facilita la lectura y garantiza que las herramientas automatizadas puedan analizar los logs con eficacia.
- Revisa y elimina regularmente los logs:** Los logs pueden contener a menudo "*log noise*", es decir, mensajes que en su día fueron útiles pero que ahora los saturan. Revisa y elimina periódicamente estos mensajes para que los registros sigan siendo un recurso valioso.

Monitorización con Actuator

Actuator es un conjunto de características adicionales que ayudar a monitorizar y gestionar las aplicaciones Spring en producción incorporando métricas y funciones de auditoria. Su utilización es muy sencilla, usando endpoints HTTP, aunque también admite JMX. Se dice que es “agnóstico” de la tecnología, lo que quiere decir, que lo podemos usar tanto sobre aplicaciones web como otro tipo de aplicaciones que permite Spring.

Para activar Actuator basta con añadir la dependencia **starter-actuator** al *pom.xml*.

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

Endpoints

Los *endpoints* nos permite monitorizar e interactuar con la aplicación. Actuator tiene muchos definidos como veremos a continuación, pero además podríamos incorporar otros creados por nosotros. Se accede a los endpoints como a cualquier otra URL de nuestra aplicación, partiendo de la raíz de la misma y seguidos del prefijo */actuator*. Un ejemplo de endpoint es */health*, el cual nos proporciona información de estado básica de la aplicación. Ejemplo: <http://localhost:9000/actuator/health>

Configuración de endpoints

Actuator viene con la mayoría de los endpoints desactivados, los únicos que por defecto vienen activados son */health* e */info* (siempre precedidos de */actuator*). Para activarlos, incluimos en el archivo *application.properties* la propiedad *management.endpoints*. Si queremos activar todos añadiremos:

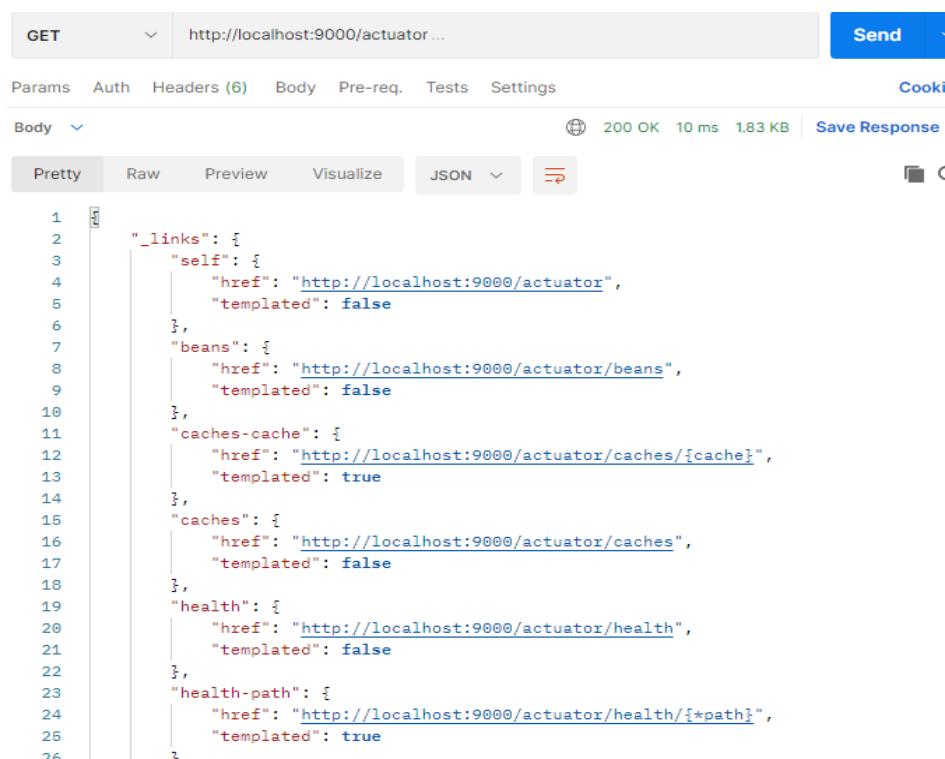
```
management.endpoints.web.exposure.include=*
```

Si queremos activar solo un endpoint específico, por ejemplo, *flyway*:

```
management.endpoint.flyway.enabled=true
```

Por otro lado, también podemos activar todos los endpoints menos uno, por ejemplo, *flyway*:

```
management.endpoints.web.exposure.include=*
management.endpoints.web.exposure.exclude=flyway
```

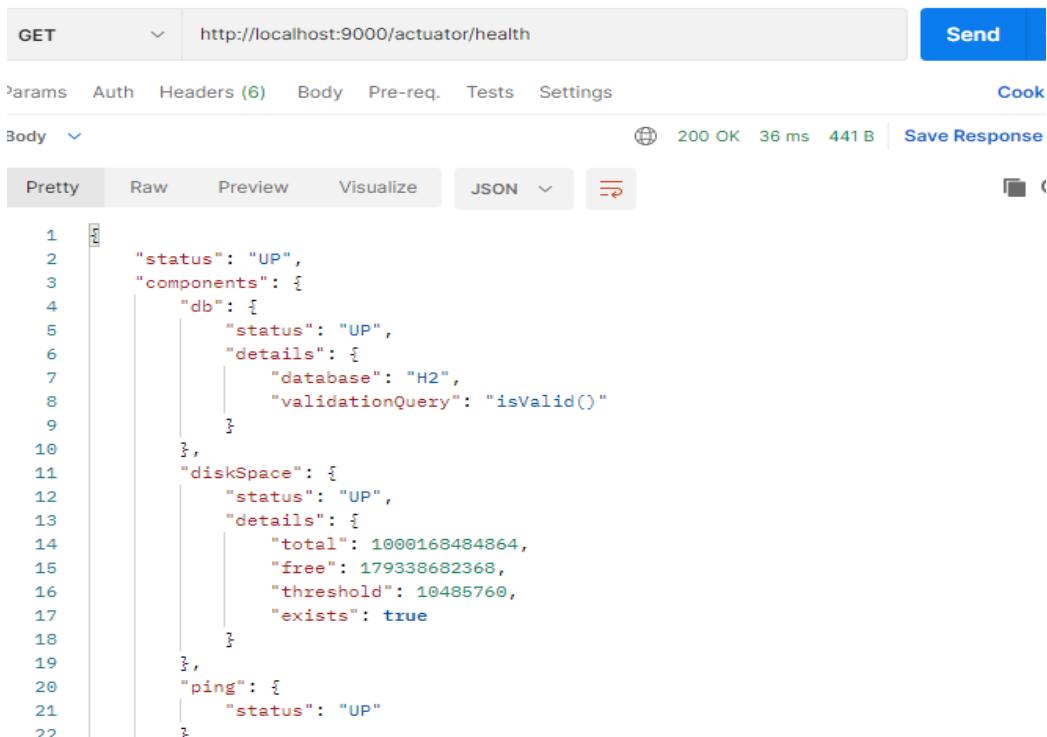


```

1  "_links": {
2      "self": {
3          "href": "http://localhost:9000/actuator",
4          "templated": false
5      },
6      "beans": {
7          "href": "http://localhost:9000/actuator/beans",
8          "templated": false
9      },
10     "caches-cache": {
11         "href": "http://localhost:9000/actuator/caches/{cache}",
12         "templated": true
13     },
14     "caches": {
15         "href": "http://localhost:9000/actuator/caches",
16         "templated": false
17     },
18     "health": {
19         "href": "http://localhost:9000/actuator/health",
20         "templated": false
21     },
22     "health-path": {
23         "href": "http://localhost:9000/actuator/health/{*path}",
24         "templated": true
25     },
26 }
```

Los endpoints ofrecen alguna configuración, que se hace desde el mismo fichero, por ejemplo, `health` permite indicarle que muestre más información mediante la propiedad `show-details`:

`management.endpoint.health.show-details=ALWAYS`



```

1
2   "status": "UP",
3   "components": {
4     "db": {
5       "status": "UP",
6       "details": {
7         "database": "H2",
8         "validationQuery": "isValid()"
9       }
10    },
11    "diskSpace": {
12      "status": "UP",
13      "details": {
14        "total": 1000168484864,
15        "free": 179338682368,
16        "threshold": 10485760,
17        "exists": true
18      }
19    },
20    "ping": {
21      "status": "UP"
22    }
}

```

Un último aspecto importante a comentar es que **deberíamos restringir el acceso a los endpoints, de forma que solo usuarios administradores** pudiesen ejecutarlos. Como todos están bajo la ruta `/actuator` es sencillo, tal y como vimos en el tema anterior, en el `SecurityFilterChain`:

`.requestMatchers("/actuator/**").hasRole("ADMIN")`

[Lista de todos los endpoints](#)

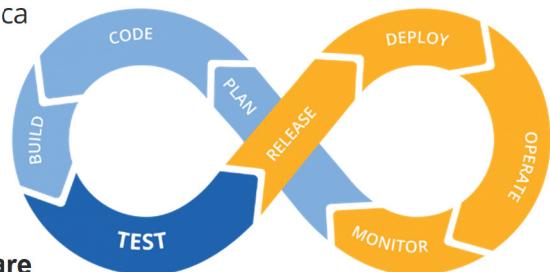
- `/auditevents`: muestra un listado de los eventos relacionados con la auditoría de seguridad, como como el login.
- `/beans`: devuelve todos los beans que se encuentran en BeanFactory.
- `/conditions`: crea un informe de las condiciones.
- `/configprops`: nos permite obtener todos los beans de `@ConfigurationProperties`.
- `/env`: devuelve el actual entorno de properties.
- `/flyway`: proporciona información sobre Flyway.
- `/health`: el estado de salud de nuestra aplicación.
- `/headdump`: devuelve el head dump de nuestra JVM que es usada por la aplicación.
- `/info`: devuelve información general.
- `/liquibase`: se comporta como el endpoint de flyway pero para Liquibase.
- `/logfile`: logs de aplicación.
- `/loggers`: nos permite consultar y modificar el nivel de logs.
- `/metrics`: devuelve las métricas de nuestra aplicación.
- `/prometheus`: devuelve las métricas formateadas para prometheus.
- `/scheduledtasks`: proporciona información sobre las tasks de la aplicación.
- `/sessions`: devuelve las HTTP Session que estamos usando en Spring Session.
- `/shutdown`: apagado de la aplicación
- `/threaddump`: vuelva la información del hilo de JVM

Despliegue (Deploying)

En este apartado podríamos hablar de herramientas de integración continua/entrega continua (CI/CD), pero sería muy amplio. Dejemos solo a modo de introducción unas definiciones y herramientas:

Integración continua: CI/CD (*continuous integration*) es una práctica por la cual los desarrolladores integran o combinan el código en un **repositorio común** (como GitHub) durante el proceso de desarrollo, no al final, facilitando la realización de pruebas y evitar conflictos entre sí.

Entrega continua: CI/CD (*continuous delivery*) consiste en la **automatización del proceso de pase a producción del software** de forma ágil y sólida. La entrega continua acelera la entrega de software a los usuarios y puede ser realizada varias veces al día, a la semana, según el proyecto.



Despliegue continuo: (*continuous deployment*) en inglés, tiene las mismas iniciales que entrega continua (CD), por lo que a veces genera confusión, porque además están muy relacionados ya que el despliegue continuo porque va un más allá de la entrega continua, automatizando todo el proceso de entrega de software al usuario, **eliminando la acción manual o intervención humana** necesaria en la entrega continua. Si alguno de estos pasos no se concluye de forma satisfactoria (por ejemplo, los tests), el despliegue no se llevará a cabo.

DevOps: es una metodología que integra a los desarrolladores de software y a los administradores de sistemas, para llevar a cabo todas las fases del desarrollo y despliegue de software.

Jenkins: es una aplicación que ayuda en la automatización de la integración continua y facilita ciertos aspectos de la entrega continua. Permite trabajar con herramientas de control de versiones como Git y ejecutar proyectos basando en Ant o Maven, así como secuencias de comandos de consola. **GitHub Actions** es una herramienta con funcionalidad similar.



Despliegue en la nube

Un aspecto importante en el despliegue es la publicación de la aplicación en un entorno web público como puede ser Amazon Web Services. Básicamente debemos crear una máquina virtual en la plataforma, la que AWS llama “instancia”, configurar el entorno de red, puertos, sistema operativo de la instancia, etc. Una vez la máquina virtual o instancia esté operativa, nos conectaremos a ella, por ejemplo, mediante una consola SSH, instalaremos Java, subiremos el archivo “jar” de nuestra aplicación y la ejecutaremos mediante el comando `java -jar`.

Con esto sería suficiente, aunque no sería la forma ideal de trabajar, ya que lo ideal es disponer de un sistema automatizado de subida de las nuevas versiones de nuestra aplicación, que incluya la también ejecución de los test incluidos en ella.

El siguiente video explica muy bien qué es AWS y como subir nuestra aplicación a la plataforma.

https://www.youtube.com/watch?v=_vOInY6SRVE

Por otra parte, AWS nos ofrece una utilidad llamada *Elastic Beanstalk* que facilita el trabajo con nuestras aplicaciones. Elastic Beanstalk es una capa de abstracción que permite configurar un entorno de ejecución que podrá contener uno o varios servidores (instancias), también una base de datos, así como otros componentes de AWS, como Elastic Load Balancer, Auto-Scaling Group, Security Group. Aquí tienes más información: <https://www.bezkoder.com/deploy-spring-boot-aws-eb/>

Docker

Docker es una plataforma de contenedores de software que permite empaquetar y distribuir aplicaciones de manera eficiente y consistente. Las aplicaciones se ejecutan en su contenedor, que está aislado, con su sistema operativo, con todo el software necesario, librerías, etc., lo que significa que se pueden mover fácilmente de un entorno de desarrollo a un entorno de producción sin tener que preocuparse por las diferencias en la configuración del sistema anfitrión.

Es un sistema similar al de máquinas virtuales similar al usado por VirtualBox o WMWare, pero con imágenes mucho más ligeras ya que no “reparten el hardware” ni disponen de un sistema operativo propio independiente, sino que el *Docker Engine*(o motor de Docker), gestiona y comparte los recursos del sistema operativo anfitrión.

Docker utiliza un sistema de imágenes para describir la configuración de un contenedor. Podemos considerar que las imágenes son “plantillas” de las máquinas que queremos desplegar y un contenedor es una imagen en ejecución. Las imágenes se crean a partir de un archivo *Dockerfile*, que es un script que define los pasos para construir una imagen. La imagen no tiene por qué ser creada desde cero, se puede descargar una imagen base de un repositorio público (el más popular es **Docker Hub**) y completarla con lo que necesitemos. Esta es la característica más potente del sistema. *Por ejemplo: si tenemos una aplicación hecha con Wordpress, podemos partir de una imagen con Linux, un servidor web Apache, PHP, el gestor de base de datos, y Wordpress y sobre ella añadir los datos nuestra aplicación y la configuración necesaria.* El proceso sería entonces:

1. Configurar la imagen mediante el *Dockerfile* partiendo de una imagen base y en la que añadiríamos la aplicación y todo el software necesario no incluido en la imagen base.
2. Crear y ejecutar una instancia de esa imagen, a la que llamaremos **contenedor**. Se descargará la imagen base, si es la primera vez que la usamos.
3. Ejecutar el contenedor desde el anfitrión con Docker instalado.

Docker permite que las aplicaciones funcionen de la misma manera en cualquier entorno, lo que reduce los problemas de compatibilidad y facilita el despliegue y escalado de las mismas.

Más información sobre uso y comandos: https://www.youtube.com/watch?v=CV_Uf3Dq-EU

Instalación

Para instalar Docker en Windows (la aplicación de escritorio se denomina Docker Desktop) debemos dirigirnos a su página oficial: www.docker.com, a la sección *Developers > Getting Started > Download*. Como requisito previo, debemos tener instalado el subsistema Linux, aunque de ello se encarga la propia instalación de Docker. En cualquier caso, los pasos previos serían:

1. Debemos tener la virtualización activada (lo comprobamos desde el Administrador de Tareas > Rendimiento > CPU).
2. En “Características de Windows” debemos tener marcadas las opciones: “Plataforma de máquina virtual” y “Subsistema de Windows para Linux”.
3. Ir al Microsoft Store e instalar Ubuntu (u otro sistema Linux).
4. Desde un terminal de comandos en modo administrador, ejecutar: `wsl --install`

Ahora ya podemos instalar Docker Desktop siguiendo los pasos del asistente. Finalizada la instalación puede ser necesario ir a: *Settings > Resources > WLS Integration > Enable Integration* y seleccionar *Ubuntu*. Adicionalmente, desde el menú lateral *Add Extensions*, podemos añadir plugins como *Portainer* y *Disk Usage* que nos facilitan la gestión de los contenedores.

Un último paso a realizar sería crear una cuenta en Docker Hub para poder acceder a su repositorio de imágenes, a partir de las cuales podremos generar nuestras propias imágenes o bien crear contenedores directamente a partir de ellas.

Trabajaremos desde una terminal de comandos, aunque la interfaz gráfica de Docker Desktop nos permite realizar gestiones básicas sobre imágenes y contenedores.

Dockerización de aplicaciones

Estos serían los pasos para desplegar una aplicación SpringBoot en Docker; esta primera aproximación será válida para cualquier aplicación que no necesite almacenamiento permanente.

1.- Obtener el archivo “jar” de la aplicación; lo podemos hacer desde la terminal de Visual Studio Code, en la carpeta raíz del proyecto, con el comando: `mvn clean package -DskipTests`

El fichero `jarse` generará en la carpeta `/target` con el nombre compuesto por el `<ArtifactId>` y `<versión>` que tengamos en el `pom.xml`.

2.- Crear la imagen Docker. Para ello escribiremos un fichero **Dockerfile** (exactamente con ese nombre, sin extensión, y en la raíz del proyecto) con las características de la misma. En nuestro caso tan solo es necesario el JDK ya el archivo jar contiene el servidor web y el resto de dependencias recursos necesarios:

```
FROM bellsoft/liberica-openjdk-alpine:17
ADD target/myapp-0.0.1-SNAPSHOT.jar /app.jar
ENTRYPOINT ["java", "-jar", "/app.jar"]
```

La primera línea indica que queremos partir de una imagen Linux con `jdk-17` procedente de Docker Hub. La segunda línea añade nuestra aplicación a la imagen, y por último se indica el comando de arranque que debe utilizar el contenedor una vez inicializado.

El siguiente paso es construir la imagen con el comando **build**:

```
docker build -t myimage .
```

Ojo: hay un punto al final que representa el directorio actual.

Podemos añadirle al nombre un “tag” que representa la versión: `myimage:latest` o `myimage:2.3`. Esta imagen ya quedará lista para ser “levantada” en uno o más contenedores, siempre que deseemos. También podremos distribuirla públicamente mediante Docker Hub. Podemos ejecutar: `docker images` para ver que se ha creado correctamente la imagen.

3.- Crear y ejecutar el contenedor a partir de la imagen mediante el comando **run**:

```
docker run --name mycontainer -itp puertoHost:puertoApp myimage:version
```

El parámetro `-it` indica que se ejecute de forma interactiva (podría ser `-d` para ejecutar en `background` y liberar la terminal). La opción `-p` hace el mapeo de puertos indicando el puerto usado en la máquina anfitrión y el puerto de la imagen Docker al que el anfitrión redirigirá las peticiones, y que es el puerto de la aplicación Spring especificado en el archivo `application.properties`.

4.- A partir de este momento podríamos acceder a nuestra aplicación dockerizada desde la máquina anfitrión mediante la URL: `http://localhost:puertoHost`.

5.- Disponemos de comandos Docker para gestionar el contenedor en ejecución: ver los detalles, pararlo o borrarlo, aunque también podemos hacerlo desde Docker Desktop:

```
docker inspect mycontainer
docker stop mycontainer (también start)
docker rm mycontainer (docker rmi imagen para borrar la imagen)
```

Volúmenes Docker

Los contenedores no tienen un espacio de almacenamiento propio por lo que el esquema mostrado en el apartado anterior es válido para aplicaciones que no guardan datos en disco o bien lo hacen sobre otro servidor externo.

Para solucionar este inconveniente/flexibilidad, Docker nos ofrece los volúmenes, que son almacenes de datos que podemos emplear en nuestros contenedores. Los pasos serían los mismos que en el caso anterior, pero con las siguientes salvedades:

1.- En el archivo Dockerfile incluiremos los ficheros y carpetas necesarios para la aplicación. El siguiente ejemplo se corresponde con un programa que tuviese una base de datos H2 almacenando sus datos en un archivo llamado *myData.mv.db* dentro de la carpeta *h2dbfiles* desde la raíz del proyecto.

```
FROM bellsoft/liberica-openjdk-alpine:17
ADD myapp-0.0.1-SNAPSHOT.jar /app.jar
RUN mkdir h2dbfiles
ADD h2dbfiles/myData.mv.db /h2dbfiles
ENTRYPOINT ["java", "-jar", "/app.jar"]
```

Y crearíamos la imagen como en el apartado anterior:

```
docker build -t myimage .
```

2.- Crear el volumen en el que se guardarán los datos:

```
docker volume create appvol (lo borraremos luego con: docker volume rm appvol)
```

3.- Crear el contenedor y ejecutarlo, indicando el volumen en el que guardará los datos

```
docker run --name mycontainer -itp 9000:9000 -v appvol:/h2dbfiles myimage
```

Podríamos probar a eliminar el contenedor y crear otro nuevo sobre el mismo volumen y veríamos que los datos se mantienen persistentes.

Servidores externos en contenedores

Un uso muy interesante de los contenedores es el mantener servidores en los que se apoyan nuestras aplicaciones, por ejemplo, el servidor de base de datos, un servidor para la autenticación, servidores de correo, etc. Así no tenemos que instalarlos y configurarlos en nuestra máquina real, simplemente lanzamos el contenedor que nos interese y de esta forma, aislamos esos procesos de nuestra máquina.

En nuestro caso, creando un contenedor MySQL evitamos todo el proceso de instalación descrito al principio de este capítulo. Para MySQL, en el repositorio Docker Hub ya tenemos una imagen disponible por lo que podríamos crear el contenedor simplemente con:

```
docker run --name mysqlcontainer -itp 3306:3306
-e MYSQL_ROOT_PASSWORD=1234 -e MYSQL_DATABASE=mydb -d mysql
```

En los parámetros `-e` le indicamos los valores para las variables de entorno definidas en la imagen, en este caso, la contraseña de *root* y el nombre de la base de datos por defecto sobre la que actuará nuestra aplicación (que se creará vacía). Estos parámetros deben coincidir con los definidos en el *application.properties* de la aplicación. El último valor de la instrucción: "mysql" es el nombre de la imagen en DockerHub, podríamos añadirle el tag con la versión, por ejemplo: `mysql:8.0-debian`

Si queremos que los datos sean persistentes necesitamos un volumen Docker para el almacenamiento de los datos y tiene que vincularse obligatoriamente a la carpeta */var/lib/mysql*, quedando así:

```
docker volume create mysqlvol
docker run --name mysqlcontainer -itp 3306:3306
-e MYSQL_ROOT_PASSWORD=1234 -e MYSQL_DATABASE=mydb
-d -v mysqlvol:/var/lib/mysql mysql
```

Los parámetros de conexión en el *application.properties* serían:

```
spring.datasource.url=jdbc:mysql://localhost:3306/mydb
spring.datasource.username=root
spring.datasource.password=1234
spring.jpa.hibernate.ddl-auto=create validate para siguientes veces.
spring.jpa.properties.hibernate.dialect=org.hibernate.dialect.MySQLDialect
```

Comunicación entre contenedores

En el caso visto en el apartado anterior, tenemos un solo contenedor, que es visible desde la máquina anfitriona. Un caso diferente es aquel en el que tenemos varios contenedores que necesitan comunicarse entre sí, por ejemplo, supongamos que tenemos un contenedor con el servidor MySQL y que nuestra aplicación esté en otro contenedor. En este caso, necesitamos establecer una red de comunicación entre ambos contenedores. Los pasos serían los siguientes:

1.- Crear la red en la que estarán conectados ambos contenedores:

```
docker network create --driver=bridge mynetwork
```

2.- Ejecutamos un contenedor con el servidor MySQL, tal y como hemos explicado previamente, creando un volumen, pero asignándolo a la red creada en el paso anterior mediante **--network**.

```
docker run --name mysqlcontainer -itp 3306:3306  
-e MYSQL_ROOT_PASSWORD=1234 -e MYSQL_DATABASE=nombreBD  
-d -v mysqlvol:/var/lib/mysql --network mynetwork mysql
```

Ojo al copiar y pegar: los saltos de línea y los guiones, pueden producir un error al ejecutarlos: "docker : invalid reference format"

3.- Una vez arrancado el contenedor, podemos conectarnos a él, abrir una terminal de comandos (shell) y ver si el servicio MySQL está corriendo. Para ello usamos el comando *exec* de Docker:

```
docker exec -it mysqlcontainer /bin/bash
```

y en la consola:

```
mysqladmin -u root -p status
```

Si estuviese parado el MySQL podríamos arrancarlo:

```
docker exec -it mysqlcontainer mysql -u root -p
```

4.- Ahora ya podemos crear el contenedor para nuestra aplicación (sin almacenamiento) como hicimos en el primer apartado, desde el Dockerfile, pero reflejando la red a la que queremos conectarlo, la misma que el servidor MySQL:

```
docker build -t myimage .  
docker run -name appcontainer -itp 80:9000 --network mynetwork myimage
```

5.- El *application.properties* de la aplicación tenemos un problema: si ponemos el nombre del contenedor donde está el MySQL, a la hora de generar el "jar" producirá un error de conexión ya que ese valor no es capaz de interpretarlo en este momento. Si ponemos *localhost* compilará correctamente (si tenemos el contenedor con MySQL ejecutándose), pero en el entorno real no tendrá sentido esa configuración ya que el MySQL se está ejecutando en otro contenedor, no en la máquina anfitriona.

Para resolverlo optaremos por una solución intermedia, asignarle una variable de entorno que se especificará al crear el contenedor, pero con un valor por defecto de *localhost*. Así, podremos generar el "jar" teniendo un contenedor MySQL activo y luego le asignaremos el nombre real del contenedor:

```
spring.datasource.url=jdbc:mysql://${MYSQL_HOST:localhost}:3306/mydb
```

y luego:

```
docker run --name mycontainer -e MYSQL_HOST=mysqlcontainer -p 80:9000  
--network mynetwork myapp
```

Docker Compose

Lo visto hasta ahora es una aproximación adecuada para manejar uno o dos contenedores, pero hemos tenido que ejecutar desde línea de comandos las operaciones de crear la red, levantar cada contenedor, etc... Para trabajar con varios o muchos contenedores, disponemos de una herramienta más avanzada llamada Docker Compose.

Docker Compose permite definir y gestionar entornos multi-contenedor, configurando contenedores, redes y volúmenes en un solo archivo, simplificando la orquestación de todo el esquema. Mientras que Dockerfile se centra en la construcción de imágenes individuales, Docker Compose se enfoca en la coordinación y despliegue conjunto de múltiples contenedores (a partir de imágenes o Dockerfiles). Aquí tenemos un ejemplo de cómo podríamos crear una red de tres contenedores, uno con MySQL, otro con phpmyadmin y otro con nuestra aplicación. El fichero está en formato 'yaml' y su nombre por defecto debe ser **docker-compose.yml** (o **.yaml**)

```

version: '3'
services:
  mysqlserver:
    image: mysql:latest
    container_name: mysqlcontainer
    environment:
      MYSQL_ROOT_PASSWORD: 1234
      MYSQL_DATABASE: mydb
    volumes:
      - mysqlvol:/var/lib/mysql
    ports:
      - "3306:3306"
    networks:
      - mynetwork
    healthcheck:
      test: ["CMD", "mysqladmin", "ping", "-h", "localhost"]
  phpmyadmin:
    image: phpmyadmin/phpmyadmin:latest
    container_name: phpmyadmincontainer
    environment:
      PMA_HOST: mysqlserver
      PMA_PORT: 3306
    ports:
      - "9001:80"
    networks:
      - mynetwork
    depends_on:
      - mysqlserver
  app:
    #build:
    #  #context: .
    #  #dockerfile: Dockerfile
    image: proy1006eimg
    container_name: appcontainer
    environment:
      MYSQL_HOST: mysqlserver
    ports:
      - "80:9000"
    networks:
      - mynetwork
    depends_on:
      - mysqlserver
networks:
  mynetwork:
    driver: bridge
volumes:
  mysqlvol:
    driver: local
    name: mysqlvol

```

Vamos a explicar su estructura paso a paso. La primera línea indica la versión de Docker Compose que estamos empleado, en este momento, la actual es la 3. Luego, el archivo tiene tres secciones distintas:

services-en la que se describe cada uno de los contenedores-, **networks**-para definir la red- y **volumes** - para definir los volúmenes o discos del entorno-.

Servicios: podemos asimilar el concepto cada servicio al de cada máquina virtual que se ejecutará en el entorno. Se debe especificar obligatoriamente la imagen (**image**:) a partir de la cual se crea, el nombre que se le asignará al contenedor (**container_name**:), el mapeo de puertos anfitrión/contenedor (**ports**:) y la red (**networks**:) Opcionalmente se le pueden añadir variables de entorno (**environment**:) y otros parámetros que detallaremos más abajo.

Networks: para definir la red y su tipo, que generalmente será de tipo *bridge*.

Volumes: para definir los volúmenes empleados en los servicios.

El archivo dispone de funcionalidad adicional, entre la que vamos a destacar:

- Cada contenedor puede establecer valores para **variables de entorno** definidas en la imagen. En nuestro caso, la imagen de *phpmyadmin*, tiene dos de ellas: **PMA_HOST** y **PMA_PORT**, para indicarle el nombre del servidor y el puerto respectivamente, donde está el gestor de base de datos MySQL con el que se comunicará. En el caso de nuestra aplicación, como ya hemos comentado, en el *application.properties* teníamos la variable **MYSQL_HOST** para indicarle el nombre del servidor.
- Crear contenedor a partir de Dockerfile. En vez de indicarle la imagen, le podemos proporcionar un archivo Dockerfile; primero creará la imagen y luego el contenedor a partir de dicha imagen. Para ello emplea las etiquetas **context**: y **dockerfile**:
- Podemos crear dependencias de ejecución, para que, hasta que un contenedor no esté terminado, no genere otro. En nuestro caso, si comenzamos la creación del contenedor de nuestra aplicación, y el contenedor MySQL no está operativo, se producirá un error. Para ello incorporamos en el servicio de la aplicación que debe esperar la etiqueta **depends_on**: y el servicio por el que espera. Además podemos incluir en cada servicio una etiqueta **healthcheck**: que, como su propio nombre indica, validará que la instalación ha terminado con éxito. En nuestro ejemplo, sobre el servicio MySQL: **test: ["CMD", "mysqladmin", "ping", "-h", "localhost"]** representa que se ejecutará un comando (cmd), y ese comando será: **mysqladmin ping**. Este comando, propio de los servidores MySQL, devuelve ok si el servidor está funcionando correctamente. El modificador del comando **mysqladmin -h** indica la máquina sobre la que ejecutar la operación.

Podemos concluir que el **despliegue del contenedor con nuestra aplicación, y el despliegue del contenedor con phpmyadmin no tendrán lugar hasta el que el contenedor con MySQL esté totalmente operativo**.

Para probar el funcionamiento de este esquema ejecutaremos:

docker-compose up -d (para pararlo emplearemos **docker-compose down**)

Ahora podremos abrir un navegador en nuestra máquina anfitrión sobre la URL: <http://localhost>. Al no indicarle ningún puerto, se hace la petición al puerto 80. Tal y como está configurado, pasará dicha petición al contenedor con nuestra aplicación, a su puerto 9000, y tal y como está configurada ésta en el archivo *application.properties*, se ejecutará normalmente. Los accesos a la base de datos están configurados en ese mismo archivo, mediante la variable de entorno **MYSQL_HOST**, que ha sido establecida también en la configuración del servidor, por lo que también funcionará correctamente.

Más información sobre Docker:

<https://www.baeldung.com/dockerizing-spring-boot-application>

<https://mikenoethiger.medium.com/how-to-deploy-a-webapp-with-docker-5149204e35f2>



Docker Cheatsheet



Docker Basic Commands		Docker Lifecycle Commands	
1. docker	To check all available Docker Commands	1. docker create	Create a new container
2. docker version	To show Docker version	2. docker run --name -it -p -e -d -v	Creates a docker container from docker image
3. docker info	Displays system wide information	3. docker pause	To pause a running container
Docker Image Commands		4. docker unpause	To unpause a running container
1. docker build	-t To build Docker Image from Dockerfile	5. docker stop	To stop a docker container
2. docker pull	-t To pull Docker Image from Docker Hub Registry	6. docker start	To start a docker container
3. docker tag	-t To add Tag to Docker Image	7. docker restart	To restart docker container
4. docker images	-t To list Docker Images	8. docker attach	Attach Terminal to Running container
5. docker push	-t To push Docker Images to	9. docker wait	Block until one or more containers stop, then print their exit codes
6. docker history	-t To show history of Docker Image	10. docker rm	To remove the Docker Container, stop it first and then remove it
7. docker inspect	-t To show complete information in JSON format	11. docker kill	To stop and remove Docker containers
8. docker save	-t To save an existing Docker Image		
9. docker import	-t Create Docker Image from Tarball		
10. docker export	-t To export existing Docker		
11. docker load	-t To load Docker Image from file or archives		
12. docker rmi	-t To remove docker images		
Docker Compose Commands		Docker Container Commands	
1. docker-compose build	To build docker compose file	1. docker start	To start a Docker container
2. docker-compose up	-d To run docker compose file	2. docker stop	To stop a running docker container
3. docker-compose ls	-d To list docker images declared inside docker compose file	3. docker restart	To restart docker container
4. docker-compose start	-d To start containers which are already created using docker compose file	4. docker pause	To pause a running container
5. docker-compose run	-d To run one one of application inside docker compose.yml	5. docker unpause	To unpause a running container
6. docker-compose rm	-d To remove docker containers from docker compose	6. docker run	Creates a docker container from docker image
7. docker-compose ps	-d To check docker container status from docker compose	7. docker ps	List all the running containers. Add the -a flag to list all the containers.
Docker Logs and Monitoring Commands		8. docker exec	To Access the shell of Docker Container
1. docker ps -a	To show running and stopped containers	9. docker logs	To view Logs for a Docker Container
2. docker logs	To show Docker container logs	10. docker rename	To rename Docker Container
3. docker events	To get all events of docker container	11. docker rm	To remove Docker container
4. docker top	To show running process in docker container	12. docker inspect	Docker container info command
5. docker stats	To check cpu, memory and network I/O usage	13. docker attach	Attach Terminal to Running container
6. docker port	To show docker containers public ports	14. docker kill	To stop and remove Docker containers
Docker Prune Commands		15. docker cp	To copy files or folders between a container and from local filesystem.
Docker Volume Commands		Docker Volume Commands	
1. docker volume create		1. docker volume create	To create docker volume
2. docker volume inspect		2. docker volume inspect	To inspect docker volume
3. docker volume rm		3. docker volume rm	To remove docker volume
Docker Networking Commands		Docker Networking Commands	
1. docker network create		1. docker network create	To create docker network
2. docker network ls		2. docker network ls	To list docker networks
3. docker network inspect		3. docker network inspect	To view network configuration details
Docker Hub Commands		Docker Hub Commands	
1. docker search		1. docker search	To search docker image
2. docker pull		2. docker pull	To pull image from docker hub
3. docker login		3. docker login	To Log in to a Docker registry
4. docker push		4. docker push	Push an image or a repository to a registry
5. docker tag		5. docker tag	Create a tag TARGET_IMAGE that refers to SOURCE_IMAGE
5. docker logout		6. docker logout	To logout from Docker Hub Registry

Tareas planificadas

La ejecución automática de programas va a permitir la realización de tareas sin intervención humana, y podemos planificarlas para que se realicen a horas específicas (por ejemplo, todos los días a las 2 a.m.), o bien cada cierto intervalo de tiempo (por ejemplo, ejecutar cada hora).

Tareas típicas de esta naturaleza son la copia de datos, volcado de información a repositorios históricos, gestión de logs. En los ejemplos que hemos visto durante el manual, podríamos planificar una tarea para cada 1 de enero se subiese el salario a todos los empleados. También nos sirven para posponer tareas no críticas, para que se realicen en horas de menos tráfico, así optimizamos el rendimiento del sistema.

A nivel de código, disponemos de una anotación `@Scheduled` que hará que el método sobre el que se aplica sea planificado. No es necesaria ninguna dependencia adicional para emplearla, simplemente la clase que la emplee deberá ser un *bean* anotado con `@EnableScheduling`.

Para tareas que deseamos que se ejecuten periódicamente, la anotación `@Scheduled` llevará el atributo `fixedDelay` con los milisegundos de intervalo. La tarea se ejecutará al arrancar la aplicación, y luego, de forma repetitiva, cada vez que transcurra el intervalo indicado.

```
@Component
@EnableScheduling
public class TareaPlanificada {

    @Scheduled (fixedDelay = 1000 * 60)
    public void tarea1() {
        System.out.println ("Ejecución cada minuto:" + LocalDateTime.now());
    }
}
```

Si queremos ejecutar algo a una hora determinada, emplearemos el atributo `cron`. El valor de este atributo es una cadena, formada por seis bloques, cada uno con una referencia temporal concreta (segundos, minutos, horas, día mes y día de la semana). A cada bloque le asignaremos el valor deseado o bien un asterisco, si queremos expresar que sea “todos” en esa referencia temporal. También puede incluir una interrogación “?” para que no se tenga en cuenta ese bloque.

Es más fácil de entender con un ejemplo:

```
@Scheduled(cron = "00 15 10 * * *")      //ejecución todos los días a la 10:15h
public void tarea2() {
    System.out.println ("Ejecución a una hora dada:" + LocalDateTime.now());
}
```

Los tres primeros bloques indican que se ejecute a las 10h, 15 min, 00 segundos. El resto de los bloques indican que se ejecute siempre, es decir, todos los días de todos los meses, cualquier día de la semana. Los valores posibles para cada bloque son, respectivamente: 0 a 59 para segundos y minutos, 0 a 23 para horas, 1 a 31 para días, 1 a 12 para meses y 0 a 6 para días de la semana, empezando el cero en lunes.

Existen otros atributos que podemos aplicar como: `fixedRate`, `initialDelay`, etc. y también podríamos usar fracciones y rangos en cada bloque del atributo. Aquí van otros ejemplos:

```
"0 0 * * *"
    // al comienzo de cada hora
"*/10 * * * *"
    // cada 10 segundos
"0 0 8-10 * * *"
    // a las 8, a las 9 y a las 10 de la mañana, cada día
"0 0 8,10 * * *"
    // a las 8 y a las 10, cada día
"0 0/30 8-10 * * *"
    // 8:00, 8:30, 9:00, 9:30 y 10:00 cada día
"0 0 9-17 * * MON-FRI"
    // cada hora de 9 a 17h, de lunes a viernes
"59 59 23 31 12 ?"
    // a la hora de las campanadas de fin de año
```

Microservicios

Una aplicación monolítica se define normalmente en un archivo .jar o .war manteniendo toda su funcionalidad concentrada y empaquetada en ese archivo. Por ejemplo, un carrito de compras en línea puede tener funciones de usuarios, catálogos de productos y las órdenes de compra.

Esta aproximación tiene ventajas e inconvenientes. Como mayor ventaja, tenemos todo el contenido junto con lo que es más fácil su comprensión, gestión y despliegue; además al no haber “piezas” no es necesario establecer un sistema de comunicación entre ellas. Como inconvenientes, es más complicado un desarrollo ágil (despliegue por partes) y, sobre todo, una asignación dinámica de recursos, es decir, poder asignarle más servidores a una funcionalidad de la aplicación sobre otra funcionalidad menos pesada.

Ventajas de los microservicios:

- Se ejecutan de manera autónoma, centrándose en una única área de negocio.
- Se despliegan y redimensionan de forma autónoma sin afectar a los demás microservicios.
- Pueden estar escritos en distintos lenguajes de programación ya que se comunican a través de API.
- Permiten una gestión ágil del ciclo de desarrollo y operación. Evitan que el mantenimiento de las aplicaciones sea costoso debido a la poca reutilización del código. Además, un fallo queda localizado en un microservicio y no tiene por qué afectar a todo el sistema.

Arquitectura de microservicios

Para trabajar con microservicios debemos descomponer la aplicación funcionalmente en diferentes APIs, siguiendo con el ejemplo del carrito de la compra, podríamos tener un componente o microservicio para la gestión de *Usuario*, otro para *Producto*, otro para *Compra*, etc. y cada uno en su archivo jar o war. La cantidad de microservicios en la que descomponemos una aplicación se conoce como “granularidad”. Una granularidad alta implica muchos microservicios que, por una parte, facilitan la escalabilidad y autonomía, pero generan más complejidad de orquestación de los mismos.

Este tipo de arquitecturas tienen una serie de componentes específicos que vamos a detallar y que cada uno se corresponderá con una aplicación independiente. En nuestro ecosistema, están amparados bajo el proyecto Spring Cloud.

- **Servicio de registro:** se encarga de “registrar” cada uno de los microservicios, para saber cuáles están activos y cuáles no, su ubicación, cómo acceder a ellos, etc. Uno de los más empleados en Spring es **Netflix Eureka**.
- **Servicio API Gateway:** es la “puerta” única a través de la cual entrarán las peticiones a nuestros microservicios. Así evitamos que los clientes de los microservicios tengan que conocer la ubicación de los mismos. Este elemento distribuirá la petición al microservicio adecuado. Recuerda que, en algunos casos, nuestros propios microservicios serán clientes de otros de nuestros microservicios. En nuestro ámbito, anteriormente se usaba *Netflix Zuul*, actualmente **Spring Cloud Gateway**.
- **Comunicación entre microservicios:** la comunicación entre los microservicios puede ser síncrona o asíncrona. En el primer caso, el proceso cliente espera por la respuesta del proceso servidor para seguir su ejecución y su implementación se basa en llamadas API, por ejemplo, mediante **RestClient** o bien mediante **OpenFeign**. En el segundo caso, se implementa un sistema de mensajería asíncrona para la gestión de esta comunicación mediante publicaciones y suscripciones en un canal de comunicación. A este sistema, el más adecuado en entornos de alto rendimiento, es lo que denominamos **broker de mensajes**, siendo los más habituales: **Apache Kafka**, **RabbitMQ** o **Redis**.

- **Balanceadores de carga:** Para distribuir el tráfico de manera equitativa entre múltiples instancias de un microservicio, se utiliza el balanceo de carga. **Ribbon**, componente de *Spring Cloud*, ofrece esta funcionalidad, permitiendo que las solicitudes se distribuyan de manera eficiente y que se gestionen las fallas de manera transparente.

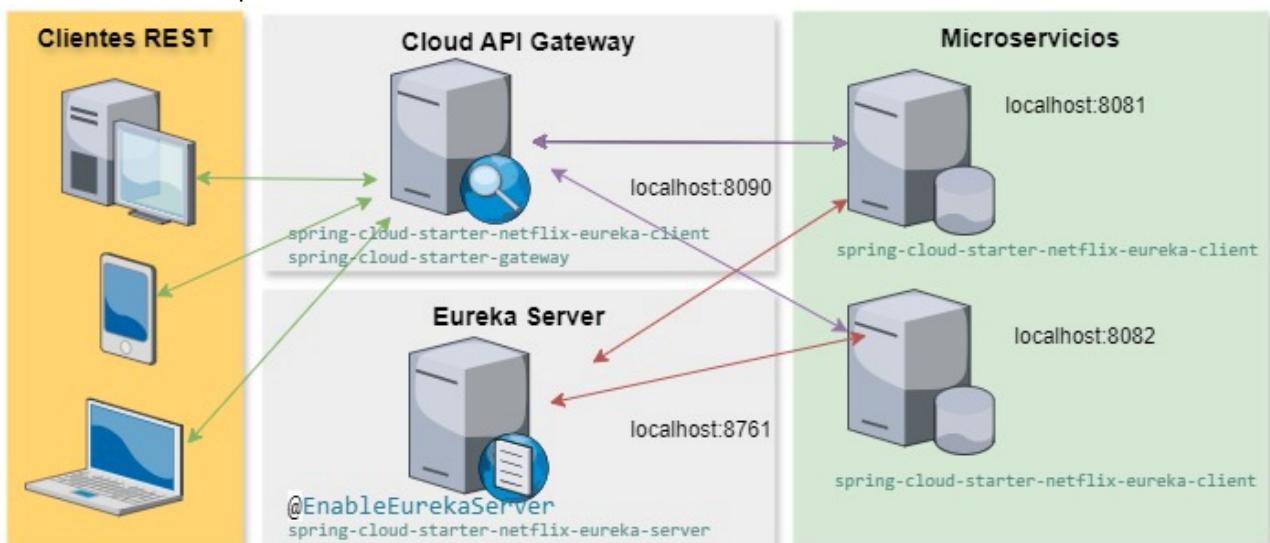
Cabe destacar que, dada la complejidad de estos sistemas, se recomienda, aunque no es obligatorio, centralizar toda la configuración de los servicios implicados en un lugar común, un repositorio accesible y modificable sin necesidad de recompilar los proyectos. Para ello disponemos de **Cloud Config Server**, un servicio que nos permitirá tener en un repositorio de Github (o similar) la configuración de todos los microservicios implicados en nuestra aplicación.

Otros componentes de este tipo de arquitecturas son las herramientas de tolerancia a fallos, herramientas de monitorización, contenedores, etc.

Proyecto SpringBoot con MicroServicios

Vamos a ver un ejemplo que incluya los componentes mínimos de un proyecto basado en microservicios, esto es: el servidor de registro *Netflix Eureka Server*, el *API Cloud Gateway* y algún microservicio registrado (Eureka client). Dejamos fuera de este ejemplo la comunicación mediante un broker de mensajes de forma que supondremos que los clientes se comunican por API mediante *RestClient*. Veremos en un apartado posterior como incorporar *Cloud Config Server* para centralizar la configuración.

También quedan fuera de este ejemplo los filtros que se pueden aplicar en el Gateway, por ejemplo, la autorización de operaciones a los usuarios.



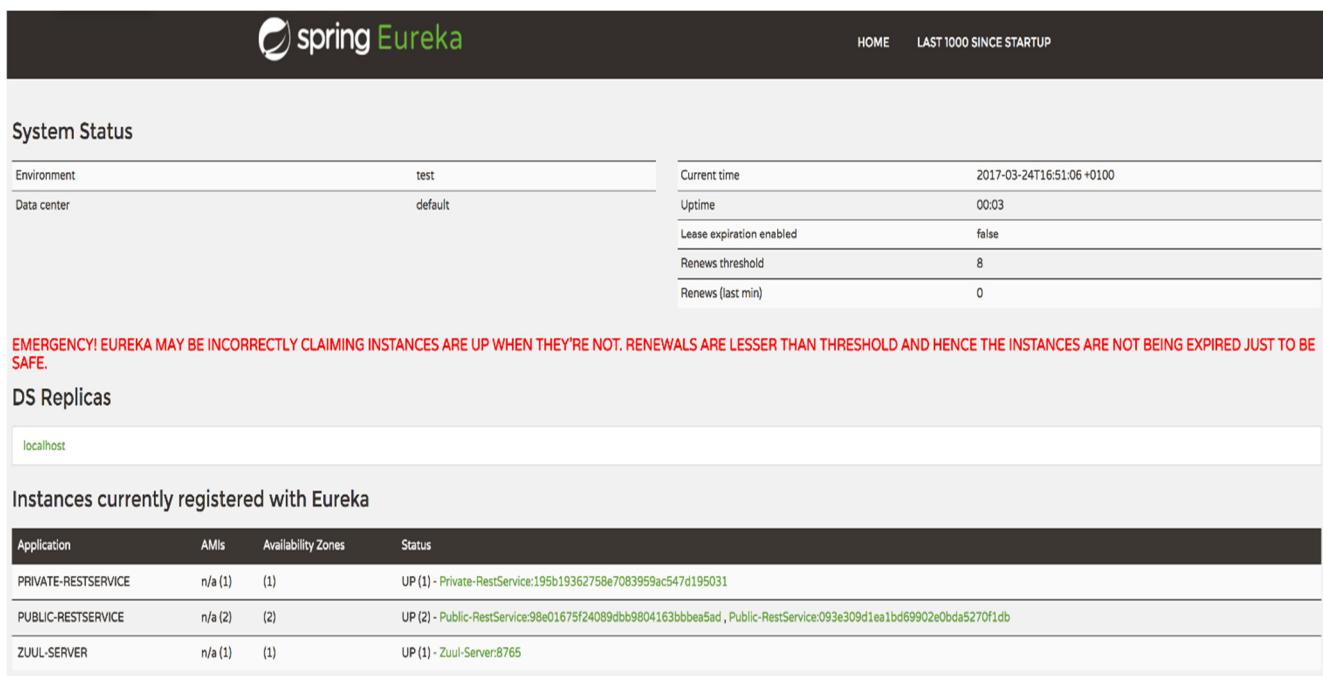
Service Registry: Netflix Eureka

El servicio Netflix Eureka será una aplicación Spring similar a las que hemos creado hasta ahora, con las siguientes características:

- Incluir la dependencia `spring-cloud-starter-netflix-eureka-server` además de `starter-web`.
- Anotar la clase Main del proyecto SpringBoot con **`@EnableEurekaServer`** para activar la autoconfiguración como servidor Eureka.
- Configurar en `application.properties` las propiedades del servidor:

```
server.port=8761
eureka.client.fetch-registry=false
eureka.register-with-eureka=false
spring.application.name=servicio-eureka-server
```

Podemos navegar la interfaz gráfica de Eureka (<http://localhost:8761>) y validar los servicios que se vayan incorporando.



The screenshot shows the Spring Eureka dashboard. At the top, there is a navigation bar with the Spring logo and the word "Eureka". Below the navigation bar, the title "System Status" is displayed. The status table shows the following data:

Environment	test	Current time	2017-03-24T16:51:06 +0100
Data center	default	Uptime	00:03
		Lease expiration enabled	false
		Renews threshold	8
		Renews (last min)	0

A red warning message at the bottom of the status table reads: "EMERGENCY! EUREKA MAY BE INCORRECTLY CLAIMING INSTANCES ARE UP WHEN THEY'RE NOT. RENEWALS ARE LESSER THAN THRESHOLD AND HENCE THE INSTANCES ARE NOT BEING EXPIRED JUST TO BE SAFE." Below the status table, the "DS Replicas" section is shown, with a single entry for "localhost". The "Instances currently registered with Eureka" section lists three services:

Application	AMIs	Availability Zones	Status
PRIVATE-RESTSERVICE	n/a (1)	(1)	UP (1) - Private-RestService:195b19362758e7083959ac547d195031
PUBLIC-RESTSERVICE	n/a (2)	(2)	UP (2) - Public-RestService:98e01675f24089dbb9804163bbbea5ad , Public-RestService:093e309d1ea1bd69902e0bda5270f1db
ZUUL-SERVER	n/a (1)	(1)	UP (1) - Zuul-Server:8765

API Cloud Gateway

Otro componente fundamental en esta arquitectura es el API Gateway, que es una “puerta” a través de la cual entrarán las peticiones a nuestra aplicación y se distribuirán a los microservicios correspondientes. Será una vez más una aplicación Spring con estas características:

- Incluir la dependencia *spring-cloud-starter-parent* (como en el proyecto anterior)
- Incluir las dependencias *spring-cloud-starter-gateway* y *spring-cloud-starter-netflix-eureka-client* además de *starter-web*.
- Registrar en *application.properties* las rutas de los microservicios que queramos exponer:

```

server:
  port: 8090
eureka:
  client:
    register-with-eureka: true
spring:
  application:
    name: gateway-service
  cloud:
    gateway:
      routes:
        - id: first
          predicates:
            - Path=/productos/**
          uri: http://localhost:8081/
        - id: second
          predicates:
            - Path=/clientes/**
          uri: http://localhost:8082/
        - id: auth-server
          predicates:
            - Path=/login/**
          uri: http://localhost:8088/

```

Microservicios (Eureka Clients)

A continuación, debemos construir nuestros microservicios, también con la siguiente configuración.

- Incluir la dependencia *spring-cloud-starter-eureka* además de *starter-web*.

- En el *application.properties* añadimos la configuración para que pueda localizar el Servicio de Registro de Eureka.

```
server.port=8081
spring.application.name=micro1
eureka.client.register-with-eureka= true
eureka.client.service-url.defaultZone=http://localhost:8761/eureka
```

- Los microservicios consumirán recursos de otros microservicios mediante RestClient, como vimos en capítulos anteriores, y ofrecerán recursos mediante sus *endpoints* (API Rest).
- La comunicación entre los microservicios se hará siempre a través de la URL del API Gateway, no la propia de cada microservicio.

Cloud Config Server

Este servicio se encarga de mantener la configuración de todos los componentes de una arquitectura de microservicios en un único lugar, en vez de que cada uno tenga la suya propia dentro de cada proyecto Spring.

Gestionará toda la configuración mediante un único repositorio en una ubicación como Github o GitLab, con las ventajas evidentes que conlleva:

- La configuración centralizada facilita los cambios en la arquitectura.
- El almacenar la configuración en un repositorio Github/GitLab incorpora la gestión de versiones.
- Al tener la configuración fuera de los proyectos, los cambios en la misma no implican modificar ni recompilar los proyectos.

Los pasos a seguir serían:

1.- Crear un repositorio en Github. Si lo hacemos mediante *push* de una carpeta local, debe estar ubicada fuera de las carpetas de los proyectos Spring. Este repositorio contendrá la configuración de cada uno de los elementos de nuestra aplicación. Cada servicio tendrá su propio archivo en el repositorio, con el nombre del servicio y extensión *.yml*.

2.- Crear el servidor de configuración. Este servidor será una aplicación Spring que tendrá anotada su clase Main con: `@EnableConfigServer`. Debe incluir también las siguientes dependencias:

```
spring-cloud-config-server
spring-cloud-starter-bootstrap
```

Este *bootstrap* no tiene nada que ver con el framework de cliente

3.- Eliminamos del servidor el archivo *application.properties* y añadimos un archivo *bootstrap.yml* que cumple la misma función. En él se indica el repositorio Github del punto 1 de donde debe tomar la configuración. Debemos incluir las credenciales Github (el usuario y el *token*). Estos *tokens* se generan desde: <https://github.com/settings/tokens>.

```
server:
  port: 8888
spring:
  application:
    name: configserver
  cloud:
    config:
      server:
        git:
          uri: https://github.com/username/cloudconfig.git
          username: username
          password: ghp_a1NLzzWB9AWag8z0gqTbd31df4dgoU3PdQQd
          default-label: main
```

Como el *name* indicado en el archivo anterior es *configserver*, en el repositorio deberemos tener un archivo *configserver.yml* de donde tomará la configuración.

4.- Para cada microservicio del que queramos externalizar la configuración deberemos realizar los siguientes cambios en los mismos:

a) Añadir las dependencias:

```
spring-cloud-starter-bootstrap  
spring-cloud-starter-config
```

b) Sustituir su *application.properties* por *bootstrap.yml* informándole solo del nombre del microservicio y la ubicación del servidor de configuración descrito en los pasos previos.

```
spring:  
  application:  
    name: microservicio1  
  cloud:  
    config:  
      uri: http://localhost:8888
```

El archivo *bootstrap.yml* podría incluir propiedades por defecto, por si no puede conectarse con el servidor de configuración.

En el repositorio Github, tendremos un archivo *microservicio1.yml* con la configuración. Ejemplo:

```
server:  
  port: 9000
```

Más información: <https://www.baeldung.com/spring-cloud-netflix-eureka>

Ejemplo completo: <https://github.com/mcwumbly/spring-boot-eureka-example>

Enlaces de interés

Ayuda en General:

- <http://acodigo.blogspot.com/p/spring.html>
- <https://www.baeldung.com/>
- <https://www.bezkoder.com/category/spring/>
- <https://www.youtube.com/@javacode503/playlists>

Seguridad:

- <https://jose-antonio-henriquez-chavarria.gitbook.io/spring-desde-0/spring-security>
- <https://www.marcobehler.com/guides/spring-security>
- <http://www.it.uc3m.es/jaf/aw/practicas/4-spring/>
- <https://docs.spring.io/spring-security/site/docs/5.6.0-M1/guides/hellomvc-javaconfig.html>

Internacionalización (i18n):

- <http://acodigo.blogspot.com/2017/03/spring-i18n-multiples-lenguajes.html>
- <https://www.baeldung.com/spring-boot-internationalization>

Cookies

- <https://reflectoring.io/spring-boot-cookies/>

AOP

- <http://acodigo.blogspot.com/2017/07/tutorial-spring-aop.html>

CRUD con C# y .NET

- <https://learn.microsoft.com/es-es/aspnet/core/tutorials/first-web-api>

Integración con VueJS

- <https://www.youtube.com/watch?v=ipcTMYv0VHI&list=PLwn-XfzoSREV93uZvo1VfsddheZvudXj9>