Desarrollo en Entorno Servidor

Ejercicios

IES Fernando Wirtz Suárez. A Coruña.
Ciclo: Desarrollo de Aplicaciones Web.
Profesor: Fernando Rodríguez Diéguez

XUNTA DE GALICIA
CONSELLERÍA DE EDUCACIÓN
E ORDENACIÓN UNIVERSITARIA

Unión Europea
Fondo Social Europeo

Índice

wirtzSpring.blogspot.com

Fernando Rodríguez Diéguez
rdf@fernandowirtz.com
Versión 2023-12-15

Entrega de prácticas

La entrega del código fuente de las prácticas de hará a través de Github. Se creará un repositorio privado otorgando permisos de lectura a la cuenta del profesor. El repositorio se llamará *cursoAWDSxxx*, siendo '*curso* ' las iniciales del curso (UDAW2, ODAW2, etc.) '*xxx*' tus iniciales. Por ejemplo: José Luís Pérez López, alumno de UDAW2, creará el repositorio: UDAW2AWDS*jpl.*

Dentro del repositorio, se creará una carpeta por cada tema. Cada una de estas carpetas se llamarán: *'xxxtt'*, siendo '*xxx*' tus iniciales y '*tt*' el número de tema. Por ejemplo: José Luís Pérez López creará para el tema 7 la capeta: *jpl07*.

En la carpeta de cada tema, cada ejercicio será un proyecto independiente en su propia subcarpeta. Esta carpeta se llamará: *'xxxttee'*, siendo 'xxx' tus iniciales, 'tt' el número de tema y 'ee' el número de ejercicio. Por ejemplo: José Luís Pérez López creará para el proyecto ejercicio 3 del tema 7 la carpeta: *jpl0703*.

En el aula virtual, para cada tarea (esto es, para cada tema) se enviará un PDF siguiendo los mismos criterios de nombre *(ejemplo: jpl07.pdf)* explicando las tareas realizadas, que deberá incluir a nivel general:

- Portada con nombre, curso, tema, etc.
- URL del repositorio GitHub de tema
- Números de página e índice
- Para cada ejercicio: enunciado, explicación de la solución aportada, con alguna captura de pantalla de las partes del código más destacables. Incluye también una captura que demuestre que el ejercicio funciona correctamente.

En caso de ser necesario subir algún otro archivo al aula virtual, se generará un archivo zip con el mismo nombre *(ejemplo: jpl07.zip)* que incluya el PDF y el resto de archivos.

Tema 1: Introducción

1.1. Realiza un pequeño estudio de los principales lenguajes de programación y frameworks de *Front-End* y *Back-End* más solicitados en las ofertas de empleo en A Coruña y en España.

1.2. Realiza un pequeño estudio de los principales IDE del mercado, específicos para lenguajes concretos o generalistas.

Ejercicios de repaso conceptos de Java SE. Realiza las aplicaciones de consola con Netbeans 14 o Visual Studio Code. Si no tienes instalado el JDK o ninguno de los dos IDE, consulta el tema siguiente.

1.3. Realiza una sencilla aplicación de consola que tenga definida una clase llamada Persona con atributos privados: dni, nombre y edad. Añádele un constructor que incluya todos los atributos, getters, setters, toString y equals y hashcode basado en el dni.

Incluye un programa que defina un ArrayList con 6 personas (puedes meter sus valores por hardcode o hacer un sencillo método para que el usuario introduzca sus valores).

Desarrolla distintos métodos en el programa anterior con las siguientes características:

- Método al que se le pasa un ArrayList de Persona y devuelve la edad del mayor.
- Método al que se le pasa un ArrayList de Persona y devuelve la edad media.
- Método al que se le pasa un ArrayList de Persona y devuelve el nombre del mayor.
- Método al que se le pasa un ArrayList de Persona y devuelve la Persona mayor.
- Método al que se le pasa un ArrayList de Persona y devuelve todos los mayores de edad.
- Método al que se le pasa un ArrayList de Persona y devuelve todos los que tienen una edad mayor o igual a la media.

En el main del programa haz llamadas a los métodos anteriores y muestra por pantalla su resultado.
Nota: a la hora de crear los métodos puedes reutilizar código de forma que unos llamen a otros y minimizar el código duplicado.

1.4. Se desea hacer la gestión de las habitaciones de un hotel. Todas las habitaciones tienen un número de habitación y un proceso de check-in y check-out. En el hotel hay tres tipos de habitaciones, aunque podría haber más en el futuro:

- 3 habitaciones Lowcost (cuesta 50 euros/día todo el año).
- 10 habitaciones dobles. Tienen una tarifa normal de 100 euros/día y un incremento del 20% si el día de salida es abril, julio o agosto.
- 5 habitaciones Suite. 200 euros/día con 20% de descuento para estancias de 10 o más días.

- Debes crear una o más clases para las habitaciones y una clase para el Hotel. La clase Hotel tendrá las 18 habitaciones en un ArrayList y las marcará inicialmente como “no ocupadas”.
- El programa tendrá un menú para hacer check-in entre las habitaciones libres, check-out entre las ocupadas y listar habitaciones libres y ocupadas.
- El check-in es común para todas las habitaciones, consiste en marcar la habitación como ocupada. El check-out consiste en marcar la habitación como libre y calcular el importe a pagar que se calculará en función del tipo de habitación y de los días de estancia (quizás sea necesario almacenar la fecha de llegada en el momento del check-in)
- Sugerencia: Para probar el programa, al hacer el check-out, puedes considerar cada día como un segundo, así, si han pasado 3 segundos, considerar 3 días.

Cuestión 1: ¿Habitación debería ser una clase abstracta o una interfaz? ¿Por qué?
Cuestión 2: ¿Es útil que el método toString () en la clase Habitación?

1.5. Se desea desarrollar un programa gestione los dispositivos domóticos de un edificio. Para ello tendremos un ArrayList que contenga en principio 3 elementos, uno para el termostato de la calefacción, otro para el ascensor y otro para el dial de la radio del hilo musical, pero en el futuro podríamos tener más elementos.

El termostato tiene una fecha de última revisión, un valor entero en grados centígrados: mínimo 15, máximo 80 y la temperatura inicial es 20. El ascensor tiene una planta en la que se encuentra, pudiendo ser desde 0 a 8. La planta inicial es la cero. El dial de radio va desde 88.0 a 104.0 avanzando de décima en décima, siendo el valor inicial 88.0.

De cada elemento (y los futuros que aparezcan) deben ser capaces de realizar las siguientes funciones:

- **subir**(), incrementa una unidad el elemento domótico. Devuelve true si la operación se realiza correctamente o false si no se puede hacer por estar al máximo.
- **bajar**(): decrementa una unidad el elemento domótico. Devuelve true si la operación se realiza correctamente o false si no se puede hacer por estar al mínimo.
- **reset**(): pone en la situación inicial el elemento domótico. No devuelve nada.
- **verEstado**(): Devuelve un String con el tipo de dispositivo y su estado actual.

Además, el termostato debe incluir un nuevo método:

- **revisar**(). Fija a la fecha de revisión a la fecha actual. No devuelve nada.

Una vez definido el sistema, crea un programa que inicie un ArrayList con una instancia de cada uno de los 3 dispositivos y luego mediante un menú nos permita hacer operaciones, primero qué operación queremos realizar (0:Salir, 1:subir un dispositivo, 2:bajar un dispositivo, 3: resetear un dispositivo, 4:revisar termostato) y luego seleccionar sobre qué elemento queremos trabajar (verificando que sea un valor entre 0 y el tamaño del ArrayList -1)

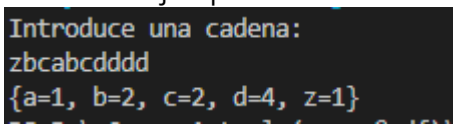
- El menú, además de las opciones nos mostrará siempre el estado de todos los dispositivos.

1.6. Realizar una clase llamada Primitiva que tenga definido una colección de 6 elementos con el resultado de un sorteo de la primitiva (serán 6 enteros con valores comprendidos entre 1 y 49 y sin repetidos). Los números se deberán mostrar ordenados ascendentemente así que decide cual es la colección que mejor se adapta a estos requisitos.

La clase dispondrá de un constructor en el que se generan y almacenen esos números al azar, también tendrá un método al que se le pase una combinación jugada como parámetro (no necesariamente ordenada) y devuelva el número de aciertos. Realiza a continuación un programa en el que el usuario introduzca boletos (6 números sin repetidos) y le diga cuantos acertó.

Realizar control de errores, tanto si el usuario introduce valores no numéricos, números repetidos o valores no comprendidos entre 1 y 49.

1.7. Realizar un programa donde el usuario introduce un String y se muestre la cantidad de veces que aparece cada letra (ordenadas alfabéticamente, no por orden de aparición). Para tener un rendimiento óptimo, se debe recorrer el String solo una vez. Elige la colección óptima para minimizar el código necesario. Ejemplo:

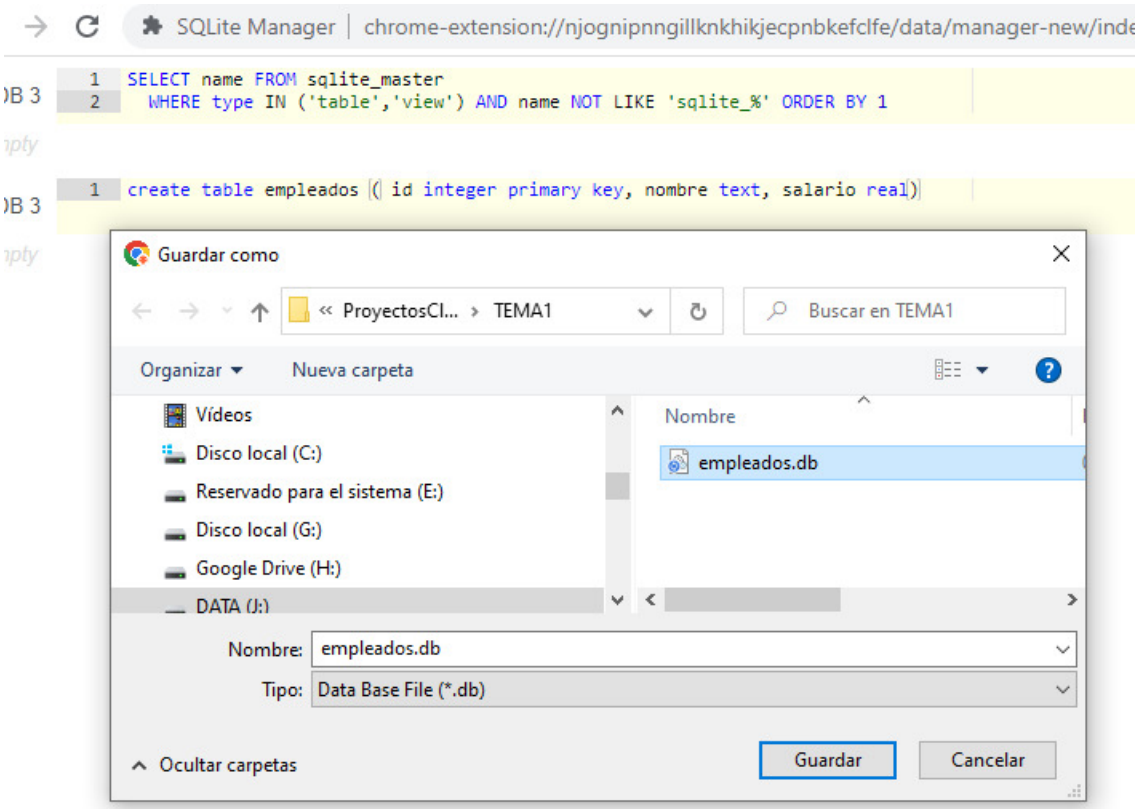


1.8. Realiza un programa de consola para la gestión de los empleados de una empresa sobre una base de datos SQLite. De los empleados mantenemos un identificador único para cada empleado, su nombre completo y su salario. El programa inicialmente cargará la tabla de empleados desde un fichero csv y luego presentará un menú para realizar las siguientes operaciones:

- **Alta**: solicita el id, nombre y salario. Opcional: puedes validar que no exista el id.
- **Baja**: solicita un id y si lo encuentra elimina el empleado correspondiente.
- **Modificación**: solicita un id y si lo encuentra, solicita nuevo nombre y salario y lo modifica.
- **Consulta**: solicita salario mínimo y máximo, y muestra los empleados con salario comprendido entre los valores introducidos.

El menú, además de las acciones anteriores, siempre mostrará el conjunto de empleados existentes en la tabla en cada momento.

El profesor te entregará un esqueleto con el código del programa (carpeta *CrudEmpleadoJdbc*) para que lo completes y también el archivo con la base de datos SQLite, aunque este último se puede crear desde el plugin de Chrome: 'Sqlite Manager'. En la siguiente imagen se ve el proceso: se crearía la tabla con el nombre que queramos (por ejemplo: empleados) y luego en el botón inferior "Save" guardamos la base de datos, que contiene esa única tabla, con el nombre que queramos. La base de datos se guardará en un solo fichero.



Cuestiones:

- Estudia qué es el patrón de diseño "Repository" y comprueba si en este ejercicio se cumple.
- ¿Qué ventajas aporta el uso de una clase BdManagerImpl y una interfaz BdManager?

Tema 2: Instalación y Primer Proyecto

2.1. Instala JDK17, Visual Studio Code y Netbeans 14. Sobre VSC instala las extensiones necesarias y sobre Netbeans instala el plugin para Spring. Realiza en ambos entornos la configuración necesaria para proyectos Java.

2.2. Crea un primer proyecto SpringBoot a través del asistente de VSC. Incluye las dependencias starter spring-Web, starter-Thymeleaf y DevTools, en Java 17, empaquetado jar. Configura para que escuche por el puerto 9000 y que solo contenga una página index.html con un titular *"Hola mundo"*. Ejecuta la aplicación y comprueba en el navegador que funciona correctamente.

2.3. Crea un segundo proyecto a partir de *https://start.spring.io* con las mismas características que el anterior. En este caso crea una estructura de páginas HTML estáticas: index, quienes-somos, productos, contacta, con un sencillo menú de navegación para las páginas y un contenido cualquiera. También sobre el puerto 9000.

Pulsa: ! + TAB al empezar a editar un documento HTML. Genera una plantilla.

2.4. Modifica el proyecto anterior para incluir alguna imagen en cada página.

Tema 3: Controladores y Vistas

3.1. Toma el proyecto del ejercicio 2.4 del tema anterior y desarrolla una clase controladora que contenga diferentes *@GetMapping* que devuelvan las vistas solicitadas (*index, quienes-somos, productos, contacta*) .

a) ¿Tienes que cambiar de ubicación las vistas? ¿Por qué?
b) ¿Tienes que cambiar el código HTML del menú de navegación de las páginas?
c) ¿Tienen que llamarse igual las rutas del *GetMapping* y las vistas?

La página index será servida para las URL*: /index*, */home*, o simplemente /. Ya que las rutas y las vistas no tienen por qué llamarse igual, renombra las vistas con el sufijo "view": *homeView.html, aboutUsView.html, productsView.html, contactView.html*.

(Opcional) Elimina el mapping *quienes-somos* y haz que muestre la vista *aboutUsView.html* mediante un archivo de configuración (implementando WebMvcConfigurer).

3.2. Añade al proyecto anterior contenido dinámico pasándole información a las plantillas mediante un *model* y representándolo con etiquetas Thymeleaf. La página de inicio puede tener el año actual, por ejemplo ©2023 tomado de la fecha del sistema del servidor. La página de *productos* puede recibir la lista con los nombres de los productos que ofrece *(por ahora puede ser un arraylist de String en el controlador, pero más adelante tomará los datos de la base de datos).*

3.3. Haz una copia del proyecto anterior y sobre él: si en la página de inicio, se le pasa el parámetro usuario=XXX mostrará el mensaje de bienvenida con un texto personalizado para ese usuario, pero si no le pasa nada, será un mensaje genérico *(Bienvenido XXX a nuestra web vs. Bienvenido a nuestra web).*
(Opcional) Haz una versión sin *Optional*, luego ponla entre comentarios y haz una segunda versión con *Optional*.

3.4. Implementa el ejemplo de los apuntes que genera números aleatorios en un nuevo proyecto. Simplemente debes crear el controlador y la plantilla con el código mostrado.

3.5. Vuelve al proyecto del ejercicio 2.2 para trabajar con el pase de parámetros a los controladores en el path. Vamos a crear un nuevo controlador, con una nueva ruta base llamada */calculos* que deberá realizar las siguientes tareas:

a) Pasa la página index.html a la carpeta /templates y crea un controlador para ella.

b) Mediante la URL: */calculos/primo?numero=X* devolverá una página diciendo que el número X es primo o no. Por ahora, haz los cálculos en un método del propio controlador, aunque en capítulos siguientes los moveremos a otras capas.

c) Modifica el apartado anterior, para que, si no introduce ningún número al que calcular si es primo o no, lo redirija a un controlador que trate el error. Este controlador por ahora simplemente mostrará la página de inicio.

d) Mediante la URL */calculos/hipotenusa/X/Y* devolverá una página con el valor de la hipotenusa correspondiente a los catetos X e Y. Si los números son negativos deberá redirigir al controlador que trate el error.

e) Mediante la URL */calculos/sinRepetidos/X* devolverá una página con X números aleatorios comprendidos entre 1 y 100, sin repetidos y ordenados ascendentemente. Obviamente X debe ser un número entero y estar comprendido entre 1 y 100, sino redirigirá a la página de error. Los mostrará en una tabla con una sola columna y cada número en una fila.

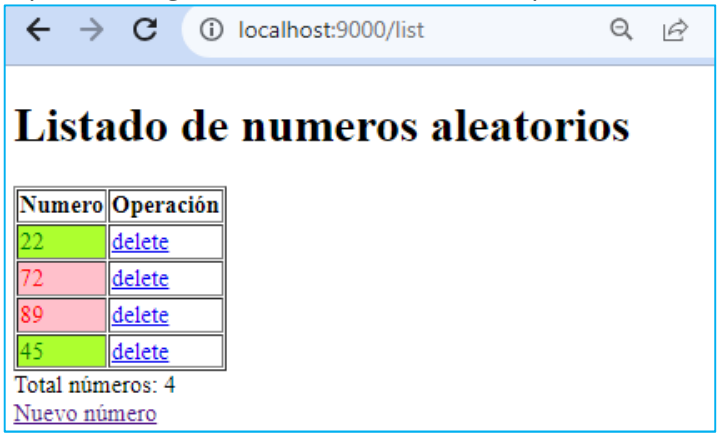
Pregunta: ¿Qué colección de Java es la más adecuada para que no contenga repetidos y estén ordenados?

f) Mediante la URL */calculos/divisores/X* devolverá una lista con divisores del número X (cada número en un párrafo). Cada elemento de esa lista será un enlace, de forma que si el usuario clica en uno de ellos mostrará los divisores del mismo. Ejemplo: para la URL */cálculos/divisores/12* mostrará 1,2,3,6, 12 y podremos clicar por ejemplo en el 6 y mostrará 1,2,3,6 y así sucesivamente.

g) Haz una versión del ejercicio anterior, pero con el parámetro X en la query de la URL, por ejemplo: */calculos/divisores?num=X*. También las URL generadas para los divisores serán enlaces con el parámetro en la query.

3.6. (Opcional) Hay una versión del ejercicio 3.4 en la que se trabaje con CSS. Los cambios a realizar serán los siguientes:
- Si la lista de números está vacía no se mostrará la tabla.
- Los números mayores de 50 se mostrarán con color de letra verde oscuro sobre fondo verde claro.
- Los números menores o iguales a 50 con color de letra rojo y fondo rosa.

Deberás crear las clases que contengan los atributos de los dos puntos anteriores.



Tema 4: Servicios

4.1. Haz una copia del proyecto del ejercicio 3.5 de temas anteriores y pasa toda la lógica de negocio (todos los cálculos) a una capa de servicio. Puede ser una sola clase (*CalculosService*).

4.2. Haz una copia del proyecto anterior y crea un área nueva para trabajar con fechas (nuevo controlador, nuevas vistas y nuevo servicio) con las siguientes características:

a) La url base de esta parte será /fechas.

b) Si le pasamos en la parte query una fecha, mostrará los días transcurridos desde el 1 de enero del mismo año. Las fechas se pasan siempre en formato YYYY-MM-DD.

c) Si le pasamos en la parte query dos fechas, mostrará los días comprendidos entre ambas fechas.

d) Si no se pasan ninguna fecha, mostrará los días transcurridos entre el 1 de enero y hoy.

e) Si pasamos /bisiesto/*fecha1* mostrará si *fecha1* pasada en el path es de un año bisiesto.

f) Si le pasamos /bisiesto/*año1*/*año2* mostrará los años bisiestos comprendidos entre ambos años.

Puedes crear en el index unos enlaces a las URL creadas con distintas fechas como parámetro y probar de forma sencilla los casos anteriores.

4.3. Haz una copia del proyecto anterior y crea una interfaz *CalculosService* pasando la clase anterior a llamarse *CalculosServiceImpl* y lo mismo con el servicio de cálculo de fechas.

4.4. Haz una copia del proyecto 3.4 pasando la lógica de negocio a una capa de servicio que contenga la colección con los números aleatorios y los métodos para añadir nuevos números, eliminar números existentes, devolver los elementos de la colección y devolver la cantidad de elementos de la colección.

Una vez que funcione la aplicación, prueba a ejecutarla en distintos navegadores a la vez. Explica en el PDF de soluciones de este tema qué ocurre, por qué y cómo solucionarlo.
Para responder a estas últimas cuestiones vuelve al tema 1 a la sección de Scopes.

Tema 5: Formularios

5.1. Haz un proyecto desde cero que contenga un formulario con dos campos de texto (nombre y edad) y un sencillo tratamiento desde un controlador *(puedes tomar el código de la captura de imagen del principio de este capítulo en los apuntes, que hace exactamente lo que se solicita)*.

5.2. Haz una copia del proyecto 4.3. y vamos a cambiar la forma de comunicaros con la aplicación. Vamos a sustituir las URL con variables en la parte query y en la parte path por formularios. Veremos como la capa de servicio permanece intacta *(así comprobamos las ventajas de la separación por capas)*. Lo haremos paso a paso:

a) Haz una página de inicio que simplemente contenga dos enlaces, uno para una página de cálculos numéricos y otro para una página de tratamiento de fechas.

b) La página de cálculos numéricos contendrá un formulario con un solo campo de texto en el que se introducirá un número. Al pulsar el botón de envío devolverá una vista indicando si el número es primo o no. Una vez que te funcione puedes, mediante las anotaciones de validación, controlar si se introduce el campo de formulario vacío o con valores no numéricos.

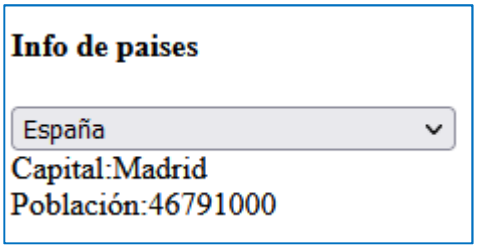
c) Añade otro formulario a la misma página, con dos campos de texto en donde introduciremos los catetos de un triángulo y un nuevo botón de envío. Al pulsarlo nos devolverá una vista con el valor de la hipotenusa.

d) Añade a la misma página un tercer formulario similar a los anteriores que muestre los divisores del número introducido. Puedes usar el mismo CommandObject que en el apartado b).

e) Vamos ahora con la página de tratamiento de fechas. Deberá contener un formulario que contenga dos cajas de texto y dos botones de radio agrupados. El primer botón de radio es para “días entre fechas” y el segundo para “años bisiestos entre fechas”. Si se selecciona el primero, al pulsar el botón de envío mostrará los días comprendidos entre las fechas introducidas en los campos de texto y si selecciona el segundo botón de radio, al pulsar el botón de envío, mostrará la cantidad de años bisiestos comprendidos entre esas fechas. Habrá que verificar que las fechas son válidas, en caso contrario redirigirá a una página de error.

5.3. Volviendo al proyecto del ejercicio 3.3 (la tienda), en la página contacto, añade un formulario con cinco campos: un texto para el nombre, un texto que debe ser de tipo email, una lista desplegable con los valores (queja, consulta, otros), una caja de texto para comentarios y un check de *“Aceptar las condiciones del servicio”*. Al enviarlo comunicar al usuario que ha sido recibido correctamente y mostrarle los datos que envió.

5.4. Realiza una aplicación con un formulario como el de la figura, en la que, al seleccionar un país en el desplegable, muestre su capital y población. Para ello habrá que realizar los siguientes pasos:



La clase de servicio deberá incluir una colección con los datos de los países (nombre, capital, población), por ejemplo:

```
private List<Pais> paises = new ArrayList<>();
```

y los métodos necesarios para su gestión:

- `cargarPaisesDesdeFichero()`:empleará el método estático *Files.readAllLines()* para pasar todas las líneas del archivo *países2019.csv* proporcionado por el profesor a un *List<String>.* Luego, aplicando el método *split()* a cada String podremos cargar la colección definida en el punto anterior. Este método se debe ejecutar al iniciar la aplicación, por lo tanto, se incluirá en un *CommandLineRunner*.

- `getPaises()`: Para pasarle al controlador (y de ahí al desplegable de la vista) todos los nombres de todos los países.

- `getPais (String nombre)` : Obtendrá los datos de un país concreto a partir de su nombre. Se invocará desde el controlador cuando se envíe el formulario, con un país seleccionado.

Este formulario es distinto a los vistos previamente ya que la página ruta destino del *submit* es la misma que la de presentación del formulario. Esto hará que, en el controlador, el GET y POST atiendan a la misma ruta.

Finalmente, para hacer más ágil la aplicación, se puede eliminar el botón de *submit* del formulario y añadir el código JavaScript necesario para que, al cambiar un valor de la lista desplegable, se envíe el formulario:

```
<select onchange="this.form.submit()" th:field="*{nombre}">
```

El formulario solo tiene un solo atributo, el nombre del país, deberíamos hacer un objeto (Command Object) con solo ese atributo pero, por comodidad, podemos hacer que el Command Object sea de tipo 'Pais'.

5.5. (Opcional) Añade al formulario del ejercicio 5.3 la posibilidad de enviar un fichero adjunto en el formulario de contacto. El fichero se guardará en el servidor en la ruta */uploadFiles*.

5.6. Toma el proyecto *MasterMind* que te entregado por el profesor y realiza las siguientes tareas:

a) Estudia el código y comenta su estructura y funcionamiento.

b) ¿Para qué incluye en el controlador y servicio la anotación *@Scope("session")*?

c) Modifica el proyecto de forma que en la página de inicio se solicite mediante un campo de texto la cantidad de intentos que tiene cada jugador para adivinar el número y mediante una lista desplegable la cantidad de dígitos que tendrá el número a adivinar.

d) Modifica el proyecto para que en la página de juego muestre la cantidad de intentos restantes y también si se produce algún error cuando el usuario introduce un intento de adivinar el número (longitud incorrecta, número con duplicados, dígitos no numéricos o número introducido previamente).

Tema 6: Modelo y Repositorios

6.1. Crea un proyecto implementando el CRUD de la entidad *Empleado* mostrado estos apuntes (sería el proyecto xxx0601). Opcionalmente, puedes crear otra versión (proyecto xxx0601b) que incluya una imagen a modo de avatar para cada empleado.

6.2. Crea un nuevo proyecto partiendo del proyecto del ejercicio 5.3 (la tienda). Sobre el nuevo proyecto crea un paquete llamado *domain* que incluya una clase llamada *Producto*. Los atributos de un producto serán: id *(long)*, nombre *(String)*, enOferta *(boolean)*, TipoIva *(enumeración: superreducido, reducido, normal)* y precio *(double)*. Crea un CRUD de Producto con un repositorio en memoria (en un ArrayList). Habría que hacer lo siguiente:

a) Estructura de carpetas/paquetes: domain, services, controllers.

b) Crear en la carpeta *domain* la clase Producto con atributos, getters, setters y constructores ayudándote de Lombok. Crear en la misma carpeta la enumeración TipoIva.

c) Crea la clase de servicio con el repositorio en memoria

d) Crear el controlador que responda a las peticiones CRUD con todos los GetMapping y los PostMapping para el envío de datos en caso de alta o modificación de Producto.

e) Crear las vistas. *El formulario de alta contendrá un botón de check para marcar si el producto está en oferta o no y botones de radio para seleccionar el tipo de IVA. El resto de campos serán cajas de texto.*

f) Crear un *commandLineRunner* que inicie el repositorio con unos valores cualquiera.

6.3. Crea un nuevo proyecto basado en el anterior para incluir una interfaz de servicio (*ProductoService*) y haz que la clase de servicio del ejercicio anterior implemente la interfaz y llámala: *ProductoServiceImplMem.*

Nota: en el tema siguiente haremos una implementación sobre base de datos en vez de en memoria.

6.4. Crea un nuevo proyecto basado en el anterior que incorpore una nueva clase llamada *Categoría*, que representará las distintas categorías de productos con id *(Long)* y nombre *(String).* Realiza el CRUD completo similar al que hiciste con Producto en el ejercicio anterior, pero sin relacionar categoría con producto por ahora.

Nota: Sería aconsejable realizar sub-carpetas para las vistas (dentro de /templates), una para las vistas de producto y otras para las de categoría.

6.5. Crea un nuevo proyecto basado en el anterior que integre ahora la relación entre el producto con su categoría. Sería de la siguiente forma:

a) Añade al Producto un nuevo atributo, que será el *id* de la categoría a la que pertenece, puedes llamarle *idCategoria.* También debemos crear en el servicio de Producto un nuevo método *findByCategory(Long idCat)* que nos devuelva los productos de una categoría, nos hará falta para mostrar los productos de una sola categoría.

b) Los formularios de alta/edición de Producto contendrá una lista desplegada para elegir la categoría.

```
<label>Categoria
 <select name="categoria" th:field="*{idCategoria}">
   <option th:each="cat : ${listaCategorias}"
     th:value="${cat.id}" th:text="${cat.nombre}">
   </option>
  </select>
 </label><br/>
```

Obviamente, ambas vistas tendrán que recibir en su model la lista de categorías.

c) La página que lista los productos incluirá una lista con las categorías de forma que al seleccionar una de ellas se filtrarán los productos, mostrando solo los de esa categoría. Para hacer esto, necesitaremos pasarle también la lista de categorías a la vista de la lista de productos.

```
public class ProductoController {
    @Autowired
    public ProductoService productoService;
    @Autowired
    public CategoriaService categoriaService;
    . . .
    @GetMapping("/")
    public String showList(Model model) {
        model.addAttribute("listaProductos", productoService.findAll());
        model.addAttribute("listaCategorias", categoriaService.findAll());
        return "product/productListView";
    }
```

y necesitamos en esa misma vista, la lista completa de categorías y el código JavaScript para que, al seleccionar una categoría, se invoque al mapping que muestre los productos solo de esa categoría:

```
<select id="select" onChange="changeCategory();">
   <option value="0">Todas</option>
   <option th:each="cat : ${listaCategorias}"
           th:value="${cat.id}" th:text="${cat.nombre}"
           th:selected="${cat.id} == ${categoriaSeleccionada.id} ? true : false">
   </option>
 </select>
 . . .
 <script>
  function changeCategory(){
   const select = document.getElementById("select");
   if (select.value == 0) window.location.href = "/";
   else window.location.href = "/porCateg/"+select.value;
  }
 </script>
```

En la lista debemos conocer la categoría seleccionada en cada momento; para ello le pasaremos desde el controlador, mediante el *model* dicha categoría seleccionada. *Podemos adicionalmente, añadir a la vista de productos un titular <h2> con esa categoría (pueden ser todas).*

```
@GetMapping("/")
public String showList(Model model) {
    model.addAttribute("listaProductos", productoService.findAll());
    model.addAttribute("listaCategorias", categoriaService.findAll());
    model.addAttribute("categoriaSeleccionada", new Categoria (0L,"Todas"));
    return "product/productListView";
}
@GetMapping("/porCateg/{idCat}")
public String showListInCategory(@PathVariable Long idCat, Model model) {
    model.addAttribute("listaProductos", productoService.findByCategory(idCat));
    model.addAttribute("listaCategorias", categoriaService.findAll());
    model.addAttribute("categoriaSeleccionada", categoriaService.findById(idCat));
    return "product/productListView";
}
```

Y en la vista opcionalmente:

```
<h2>Categoría: <span th:text="${categoriaSeleccionada.nombre}">default</h2>
```

Quedaría una última cuestión por resolver: ¿Qué hacer cuando borramos una categoría? ¿No permitimos que se borre una categoría si tiene productos asignados? ¿Se deberían borran "en cascada" todos los productos de esa categoría? ¿Dejamos los productos "huérfanos", sin categoría asignada?
La opción más sencilla sería la primera, inyectando el servicio de producto en el controlador de categorías (o mejor en servicio) y verificaríamos que no hay productos en esa categoría:

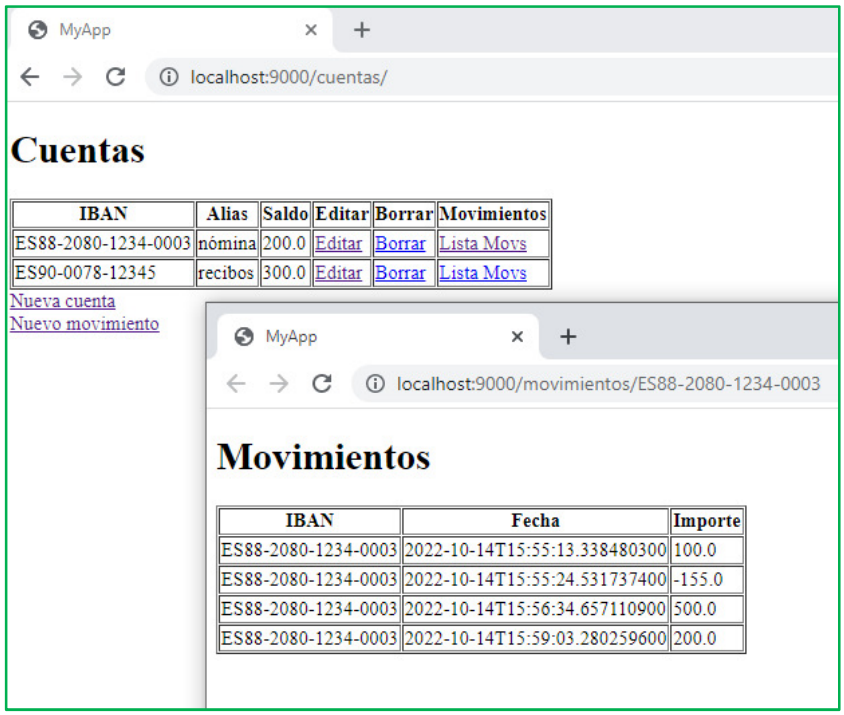
```
@GetMapping("/delete/{id}")
    public String showDelete(@PathVariable long id) {
        if (productoService.findByCategory(id).size() == 0)
            categoriaService.delete(id);
        return "redirect:/categorias/";
    }
```

La segunda opción, la de borrar en cascada, la veremos en el ejercicio siguiente, en la que, al borrar una cuenta corriente, deberemos borrar automáticamente todos sus movimientos.

6.6. Crea un proyecto que gestione las cuentas corrientes de una empresa y sus movimientos. Una cuenta corriente se caracteriza por un IBAN, un alias o nombre, su saldo actual y un histórico de movimientos. De cada movimiento de cada cuenta tenemos una fecha/hora del mismo (que se tomará del sistema) y un importe que puede ser negativo (reduce el saldo de la cuenta) o positivo (aumenta el saldo de la cuenta). La aplicación debe permitir el CRUD de cuentas corrientes y alta y consulta de sus movimientos (una vez dado de alta un movimiento ya no se puede ni eliminar ni modificar).
Consideraciones:

- Se hará una gestión con repositorios en memoria (un ArrayList para las cuentas y otro para los movimientos).
- No se admiten ingresos de más de 1000 euros ni retiradas de más de 300 euros.
- Para el alta de una cuenta hay que informar de todos sus atributos salvo el saldo que es cero.
- Para borrar una cuenta su saldo debe estar a cero y se borrarán automáticamente todos sus movimientos. Para borrarlos no se puede hacer con un for...each, hay que emplear Iterator:

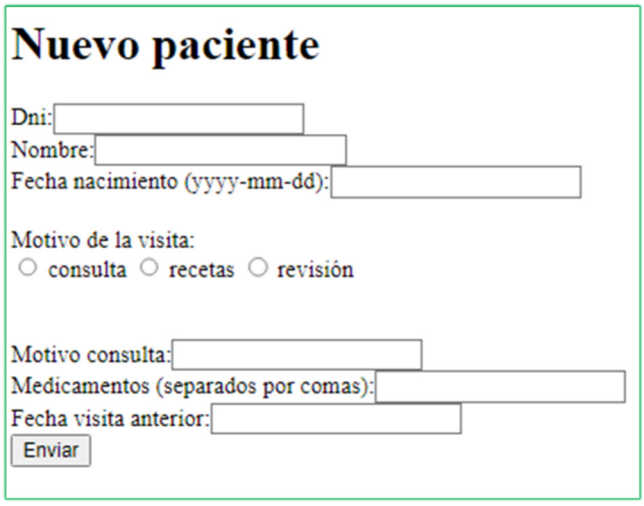
```
Iterator<Movimiento> iterator = repositorioMovimientos.iterator();
while (iterator.hasNext()) {
    Movimiento movimiento = iterator.next();
    if (movimiento.getIban().equals(iban))
        iterator.remove();
}
```



6.7. Vamos a trabajar ahora con herencia y con la lectura de parámetros desde un archivo. Se trata de hacer una aplicación que gestione la sala de espera de una consulta médica mediante un repositorio en memoria (ArrayList, LinkedList). Los requisitos son los siguientes.

- Cuando llega un paciente (en se le solicitan sus datos y pone al final de la lista (añadir al ArrayList). La vista principal de la aplicación mostrará en una tabla el nombre, DNI y fecha de nacimiento de los pacientes en la sala de espera, ordenados por orden de llegada.
- Cuando se llama a consulta a un paciente, se le calcula lo que tiene que pagar y se saca de la lista de espera. Se llama a consulta siempre el primer paciente de la lista, no se elige (es una cola FIFO).

- De todos los pacientes queremos saber: nombre, dni y fecha de nacimiento. Hay tres tipos de pacientes: los que van a consulta (queremos saber el motivo de consulta), los que van por recetas (queremos saber la lista de recetas), y los que van a revisión (queremos saber la fecha de la última revisión).
- Para simplificar haremos un único formulario con todos los datos del apartado anterior (y un botón de radio seleccionando el tipo de paciente). Obviamente no hay que cubrir todos los datos del formulario, por ejemplo, un paciente que va a consulta, no se cubrirá la lista de medicamentos. También podemos hacer que la lista de medicamentos sea una caja de texto y que el operario los introduzca separados por comas.



- Todos los tipos de paciente (y los nuevos que puedan surgir) deben tener un método que se llame *facturar,* aunque su fórmula depende del tipo de paciente: para consulta es una tarifa fija (100€), para medicamentos es un importe por cada medicamento (5€) y para revisión es una tarifa fija si es jubilado (30€) y otra si no lo es (50€).
- Estas cantidades están almacenadas en un fichero de tipo *properties* que se leerá al principio del programa. El método *facturar* recibirá como parámetro una instancia de la clase que contenga estos valores de las tarifas.
- La vista principal tendrá entonces: la lista de pacientes, un enlace o botón para añadir un nuevo paciente al final de la lista (invocando a un formulario de solicitud de datos) y un enlace o botón para llamar y facturar al primero de la lista.
- El importe a pagar por el paciente que se llama a consulta puede ser una etiqueta en esa misma vista principal que solo sea visible cuando se llama a un paciente.

  - Pista: La aplicación de debe tener las clases: Paciente (abstracta) y sus tres clases hijas. Una clase para gestionar las tarifas y una clase adicional para recibir todos los campos del formulario.
  - Pista: el servicio debe contener un método que, a partir de los datos recibidos del formulario, cree el paciente del tipo correspondiente y cubra todos sus datos.
  - Pista, si en la clase padre tenemos @EqualsAndHashCode, en las clases hijas debemos incluir: `@EqualsAndHashCode(callSuper = true)`

Video solución del ejercicio: *https://youtu.be/BIj7VHtHOiw*

Tema 7: Acceso a datos

7.1. Crea un proyecto que implemente el CRUD de Empleado partiendo del proyecto del tema anterior (ejercicio 6.1) y convirtiéndolo en persistente siguiendo los pasos indicados en el tema 7. Además de los métodos de JPARepository, incorpora los métodos derivados por nombre y los métodos *@Query* mostrados en los apuntes.

7.2. Haz un nuevo proyecto con un CRUD de Empleado (puedes partir del ejercicio anterior), añade el CRUD de Departamento e incorpora la asociación Empleado-Departamento (@ManyToOne) mostrada en los apuntes. Si no lo has hecho aún, deberías tener dividido en paquetes el proyecto: domain, controllers, services, repositories.

7.2b. Haz una segunda versión del proyecto anterior en la que se pueda elegir un departamento en la vista y muestre solo los empleados de dicho departamento.

El proceso es similar al explicado en el tema anterior para Producto y Categoria: la lista desplegada/desplegable de categorías debe incluir un evento *onChange* que nos dirija al controlador */porDepto/{idDepto}*. Esto nos obligará a crear en el servicio de empleado el método *obtenerPorDepto* que a su vez nos lleva a definir el método derivado por nombre *findByDepartamento* en el repositorio. El código JavaScript podría ser así:

```
<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:

```
@GetMapping("/")
    public String showList(Model model) {
        model.addAttribute("listaEmpleados", empleadoService.obtenerTodos());
        model.addAttribute("listaDepartamentos", departamentoService.obtenerTodos());
        model.addAttribute("deptoSeleccionado", 0); // new Departamento(0L, "Todos"));
        return "empleado/listView";
    }
```

7.3. Haz un nuevo proyecto, partiendo del anterior e incorpórale la asociación con Categoría mostrada en los apuntes (@OneToMany) de forma bidireccional. *Recuerda que en la clase Empleado debes añadir la anotación @ToString.Exclude al atributo de Categoría.*

Crea una interfaz genérica para todos los servicios como la mostrada en los apuntes (HelperService) y haz que todos los servicios la implementen.

7.4. Haz un nuevo proyecto, partiendo del anterior e incorpórale la entidad Proyecto y una relación entre Empleado y Proyecto de tipo *muchos a muchos*, (no bidireccional) con atributos extra, tal y como se muestra en los apuntes. *Para que sea más sencillo puedes optar por la opción de añadir un id a la clase asociación y no @IdClass.*

7.5. (Opcional) Haz un nuevo proyecto partiendo de cero con la entidad Empleado (atributos id y nombre) la entidad Coche (atributos id, matrícula y modelo) y añadiendo la asociación 1 a 1 entre ellas mostrada en los apuntes:
Notas:

- En el alta de empleado no se le asignará ningún coche. En la vista de empleados tampoco se muestran los coches que tienen asignados los empleados.
- En la vista de coches habrá una columna para el empleado asignado, que puede mostrar el nombre del empleado asignando o bien en blanco.
- En el alta de coche debemos validar que la matricula introducida no la tiene ningún otro coche. Además, podremos no asignarle ningún empleado o bien asignarle un empleado de los existentes, pero asegurándonos de que ese empleado no tiene ningún otro coche asignado.
- Al editar un coche, debemos verificar lo mismo que en el alta de coche, en cuanto a matrícula única y empleado no asignado a otro coche.
- Asegúrate de que no hay borrado en cascada y que solo se hace el borrado de un empleado si no tiene coche asignado.

7.6. Haz un nuevo proyecto, partiendo del ejercicio 7.4, pero que sea persistente en disco y no admita cambios en el esquema de base de datos.

- Primero debes ejecutarlo una vez sobre disco, que genere el esquema y haciendo la carga inicial de datos que tuvieses (CommandLine Runner).
- Luego debes revisar que el esquema está correctamente creado y configurar la aplicación para que no recree el esquema nunca más ni haga la carga inicial.

7.7. Crea un proyecto con una tabla de empleados (puedes partir del ejercicio 7.1) pero simplificado (solo atributos: id, nombre, email y salario) y que los muestre paginados, de 10 en 10, y que tengamos un enlace para página siguiente y página anterior. Será una aplicación solo para mostrar datos, no un CRUD completo.

- Para crear los datos de ejemplo, visita la página *mockaroo.com* y genera un conjunto de 100 inserts para la tabla de empleado. Si en la tabla, en la anotación *@GeneratedValue,* incluimos el modificador:
  `@GeneratedValue(strategy = GenerationType.IDENTITY)`
  podremos generar los inserts sin incluir el campo id. Ejemplo:
  `insert into Empleado (nombre,email,salario) values ('J.Pérez', 'jp@gmail.com', 1240);`

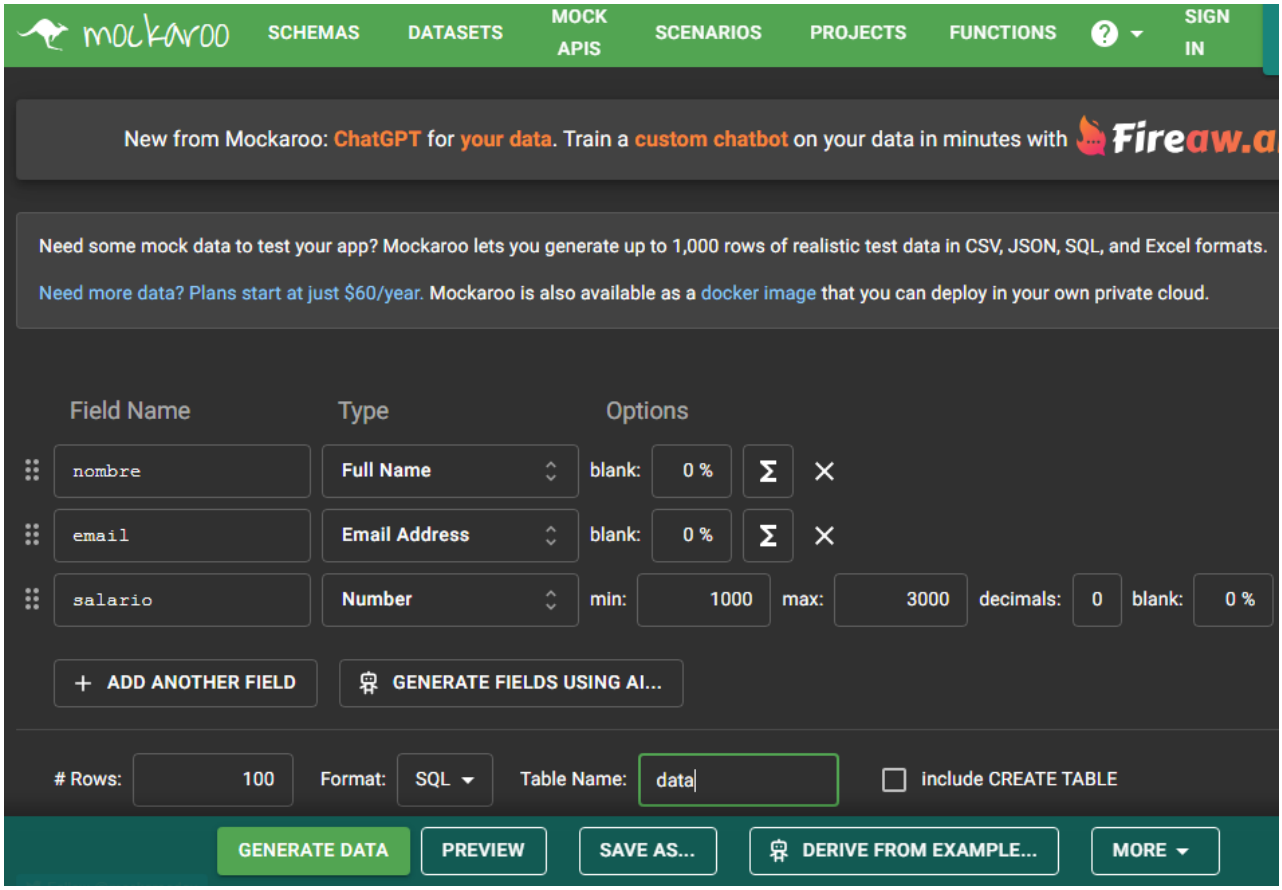
- Debes incorporar esos inserts en el fichero */resources/data.sql*. Así no es necesario tener en el servicio y controlador los métodos de inserción, borrado, etc. Solo el de mostrar los datos paginados.

- Debes incluir en el archivo *application.properties* las líneas:

```
spring.sql.init.mode=always
spring.jpa.defer-datasource-initialization=true
```

para permitir la ejecución de scripts y que cree las tablas antes de que realice los inserts del fichero *data.sql* respectivamente.

- En la vista habrá botones de navegación a página siguiente, anterior, primera y última. Para simplificar el caso de la primera y última página, en el caso de estar en la primera página, el enlace de página anterior apuntará a esa misma página. Análogamente, si estamos en la última página, el enlace a página siguiente también apuntará a esa última página de nuevo.

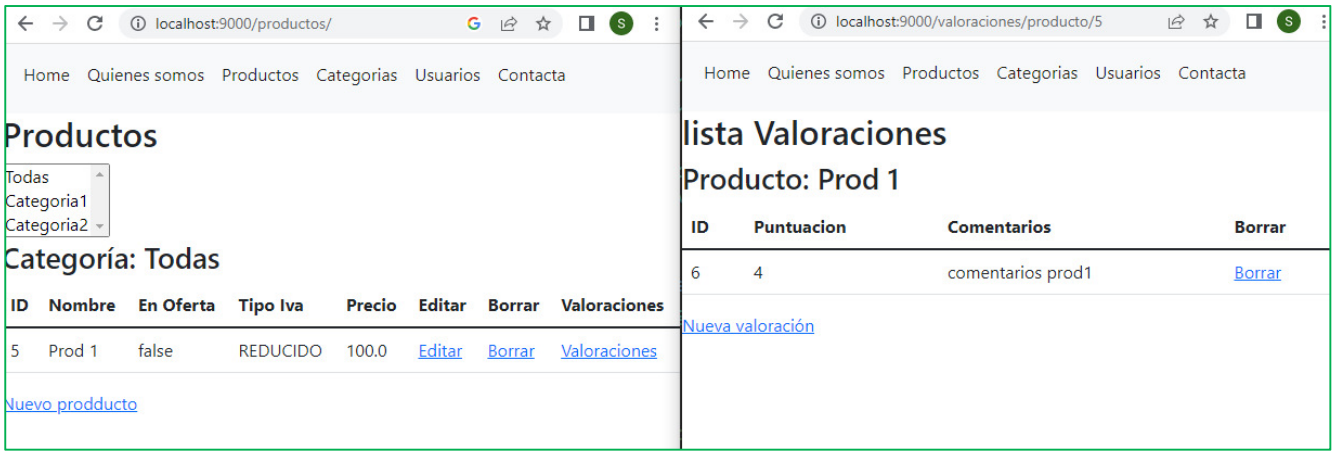


7.8. Toma el proyecto 7.2 (Empleado y su asociación con Departamento) y crea un DTO para Empleado que contenga solo id, nombre y nombre de departamento. Sustituye la lista de empleados que se le envía a la vista por una lista que contenga solo los atributos del DTO creado para empleado.
Deberás usar un ModelMapper para obtener la lista de EmpleadoDTO que pasarás a la vista, a partir de la lista de empleados completa que devuelve el servicio.

7.9. Toma el proyecto de la tienda del tema anterior (ejercicio 6.5) y convierte los repositorios de Producto y Categoría en memoria en repositorios sobre base de datos (H2 en memoria).

- Añade los parámetros de BD a *application.properties*
- Añade a las clases del dominio: *@Entity,@Id,@GeneratedValue* y las relaciones entre ellas.
- Crea los repositorios hijos de JPARepository. En el repositorio de Producto quizás sea necesario un método para obtener los productos de una categoría, no lo programes, usa los métodos derivados del nombre.
- Adapta los servicios para que llamen a los métodos de los nuevos repositorios.
- En los formularios, el campo de la relación con Categoría debe ser la clase (categoría) y no el id de la misma, como hacíamos en el ejercicio 6.5.

7.10. Añade al proyecto anterior una entidad nueva llamada 'Usuario' con atributos (id, nombre, fecha de registro). Deberás añadir una nueva asociación ya que los productos podrán ser valorados por los usuarios. De cada valoración, además del usuario y el producto, guardaremos la puntuación y un texto de comentarios.



- Generar todo lo necesario para Usuario: dominio, repositorio, servicio CRUD, controlador CRUD y vistas.
- Generar para Valoración: dominio, repositorio, servicio CRUD. En el repositorio quizás sea necesario un método para obtener las valoraciones por producto y otro para las valoraciones por usuario, no los programes, usa los métodos derivados del nombre.
- A diferencia de los otros controladores, el controlador de las valoraciones tendrá un mapping para mostrar todas las valoraciones de un producto, o todas las de un usuario, pero no uno que muestre todas las valoraciones del sistema. Por ejemplo: @GetMapping ("/valoraciones/producto/{idProd}")
- Establecer las relaciones entre las clases: Usuario – Producto - Valoración: sería una "many to many" con atributos extra. En este caso no crearemos @IdClass, lo haremos con un @Id para cada valoración.
- Adaptar las vistas de producto y usuario para poder gestionar sus valoraciones. No haremos una vista que muestre todas las valoraciones de todos los productos, se accederá a las valoraciones o bien desde un producto o desde un usuario. Por simplificar, no hacer la edición de valoraciones, solo consulta, alta y borrado.

7.11. Toma el proyecto de los movimientos de cuentas del tema anterior (ejercicio 6.6) y convierte los repositorios en memoria en repositorios sobre una base de datos H2 en memoria.

- Si la clase Movimiento no tenía asignado un id, es más cómodo que le asignes uno con @Id.
- Establece una relación unidireccional entre Cuenta y Movimiento.
- En el ejercicio 6.6 teníamos que borrar los movimientos al borrar una cuenta, ahora lo hará automáticamente al añadir *@OnDelete(action=OnDeleteAction.CASCADE)* a la entidad *Movimiento.*
- El repositorio de Movimiento necesitará el método derivado:
  List<Movimiento> findByCuenta(Cuenta cuenta);

7.11b. (Opcional) Haz una versión del ejercicio anterior con la relación bidireccional entre ambas entidades. Al incluir el ArrayList de Movimientos en la entidad Cuenta, hay ciertas operaciones que podremos hacer de forma más sencilla. El ejemplo más claro es cuando, partiendo de una cuenta determinada, queremos pasarle a una vista todos sus movimientos; con una relación unidireccional necesitaremos crear un método *findByCuenta* en el repositorio de Movimientos y llamarlo con esta cuenta como parámetro; por el contrario con la relación bidireccional, basta con hacer: *cuenta.getMovimientos().*

7.12. Toma el proyecto de la lista de espera de la clínica del tema anterior (ejercicio 6.7) y convierte los repositorios en memoria en repositorios sobre una base de datos H2. Prueba las distintas estrategias de almacenamiento de la herencia (single table / joined) y haz capturas de las tablas generadas en cada caso.

- Al pasar a BD y no un ArrayList, no sabemos cuál es el primer paciente. La forma más sencilla de solucionarlo es añadir un 'id' autogenerado para cada paciente y crear un método @Query en el repositorio para obtener el primero, esto es, el mínimo:

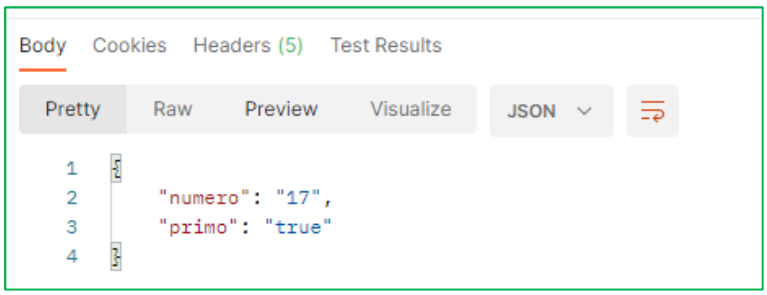
  @Query("select p from Paciente p where p.id=(select min(p2.id) from Paciente p2)")
  Paciente getFirst();
- Añadimos @DiscriminatorColumn y @DiscriminatorValue a las entidades en caso de estrategia Single Table.
- En vez de guardar en la tabla el ArrayList de medicamentos, para los pacientes que vienen por recetas tendremos solo la cantidad de medicamentos y no su nombre.

Tema 8: API Rest

8.1. Implementa el CRUD de Empleado partiendo del proyecto del tema anterior (ejercicio 7.1) y convirtiéndolo en una API REST sin gestión de excepciones.

8.2. Crea un nuevo proyecto convirtiendo el proyecto del ejercicio 4.3 (Cálculos matemáticos y fechas) a API Rest sin gestión de excepciones.

- Como queremos los valores de respuesta lleguen al cliente en formato JSON debes crear un Map que conformará el cuerpo de la respuesta.



8.3. Haz un nuevo proyecto, partiendo del 8.1 e incorpórale la asociación con Departamento (@ManyToOne) mostrada en los apuntes, creando con ModelMapper los dos DTO allí también mencionados *(uno para enviar los datos de empleados al cliente mostrando solo id, nombre y email, y otro DTO para la inserción y edición de empleados).* Puedes basarte también en el ejercicio 7.2. para la entidad y repositorio de Departamento.

8.4. Haz un nuevo proyecto, partiendo del ejercicio 8.1 (CRUD de Empleados) incorporando una gestión de errores basada en el modelo *ResponseStatusException.*

- Crea las excepciones 'empleado no encontrado' y 'base de datos sin empleados' e invócalas desde la capa de servicio. La primera se lanzaría en los métodos *findById* y *delete* y la segunda en el método *findAll*.
- En el controlador, invocar a los métodos de servicio dentro de un *try... catch*, y que en caso de que se produzca enviar al cliente un *ResponseStatusException*.
- Configura los dos parámetros en *application.properties* para que funcione *ResponseStatutException.*

8.5. Crea un nuevo proyecto convirtiendo el proyecto del tema anterior 7.11 (Cuentas Corrientes) a API Rest con una gestión de errores basada en el modelo *ResponseStatusException.*

- Por hacerlo más simple, no es necesario hacer el mapping de edición de cuentas, solo: listar cuentas, nueva cuenta y borrar cuenta, y por otra parte: listar movimientos de una cuenta y añadir movimiento.
- Ya que la asociación entre Cuenta y Movimiento es bidireccional, recuerda añadir @JsonIgnore en el atributo List<Movimiento> de la clase Cuenta, para evitar bucles infinitos al recuperar cuentas con movimientos.
- Crea un DTO para nueva Cuenta (el cliente solo envía Iban y alias ya que el saldo será cero y la lista de movimientos *null*) y un DTO para nuevo Movimiento (el cliente solo envía Iban e importe ya que el Id lo genera Hibernate y la fecha se toma del sistema). Deberás crear un método *convertDtoToCuenta* y *convertDtoToMovimiento* (puede ser en los propios servicios *CuentaService* y *MovimientoService*).

- El repositorio de movimientos puede incluir un método derivado por nombre: *findByCuenta.*
- Puedes definir 5 excepciones de servicio: Cuenta no encontrada, Base de datos de cuentas vacía, la cuenta a borrar no tiene saldo cero, movimiento con importe incorrecto, base de datos de movimientos vacía.
- Añade a los controladores *try...catch* que capturen las excepciones de servicio y llamen a ResponseStatusException.

8.6. Crea un nuevo proyecto igual al ejercicio 8.4 (CRUD Empleado con gestión de errores con *ResponseStatusException*) pero con una gestión de errores basada en *@RestControllerAdvice*.

- Solo tienes que eliminar los try...catch del controlador y añadir la clase que gestiona de forma centralizada todas las excepciones (no son necesarios los dos parámetros en *application.properties* del ejercicio 8.4)
- Comprueba con Postman que funciona tanto para las dos excepciones definidas como para una excepción genérica (por ejemplo, una url mal formada con un texto en vez de un número de empleado: GET /empleado/aaa).

8.7. Crea un nuevo proyecto convirtiendo el proyecto del tema anterior 7.10 (Tienda con Productos, Categorías, Usuarios y Valoraciones) a API Rest con una gestión de errores basada en *@RestControllerAdvice*.

- Puedes hacer que todas las rutas partan de la base /api/ conservando todo el modelo MVC del tema anterior. En este caso, si necesitamos mantener en pom.xml las dependencias thymeleaf y webjars. Conserva los controllers en la carpeta *'controllers'* y crea una nueva *'restcontrollers'* para los nuevos.
- Los mappings de 'inicio', quienes somos', etc. no es necesario implementarlos con API Rest ya que no deberían contener datos dinámicos, podrían ser estáticos de la aplicación cliente. Sí "restificaremos" el formulario de contacto.
- Por simplificar, implementa solo las excepciones 'no encontrado' de las cuatro entidades del proyecto.

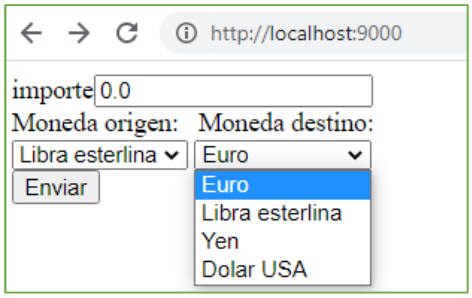
8.8. (Opcional) Crea un nuevo proyecto basado en el del ejercicio 8.5 (Cuentas y Movimientos), añadiendo HATEOAS, de forma que la respuesta a una cuenta incluya un enlace a sí misma (self) y otro a la URL que devuelve una respuesta con todas las cuentas *(repositorio.findAll)*

8.9. (Opcional) Crea un nuevo proyecto partiendo del ejercicio 8.3 pero empleando el modelo Spring Data Rest. Debes eliminar los servicios y los controladores ya que Data Rest se encarga de esa tarea. Comprueba que las respuestas cumplen el estándar HATEOAS.

8.10. Crea un nuevo proyecto a partir del proyecto 8.7 (Tienda) y documenta la API mediante Swagger: tanto documentación general: descripción, autor, etc. como información de cada método: descripción, valores devueltos y parámetros necesarios.

Para todos los proyectos anteriores deberías configurar CORS para que puedan ser consumidos desde una aplicación cliente, escrita por ejemplo en JavaScript.

8.11. Crea un proyecto Spring MVC desde cero, que contenga una vista como la que se muestra en la figura. Cuando se envíe el formulario, se consultará la cotización de la moneda origen-moneda destino y se mostrará en otra vista el importe resultante de aplicar el cambio sobre el importe introducido. Para obtener la cotización se empleará la API externa: *https://api.frankfurter.app/*, con una URL que deberá tener un formato así:

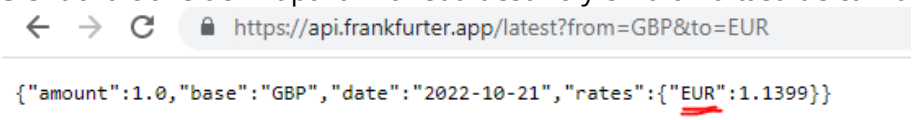


*https://api.frankfurter.app/latest?from=**XXX**&to=**YYY***. Las posibles monedas origen *(XXX)* y destino *(YYY)* serán: EUR, GBP, JPY, USD. Ejemplo*: https://api.frankfurter.app/latest?from=GBP&to=EUR*

La clase que devuelve la API tiene una estructura así:

```
import lombok.Getter; import lombok.Setter;

@Getter
@Setter
public class CambioData {
    private float amount;
    private String base;
    private String date;
    private HashMap<String, Float> rates;
}
```

Siendo la clave del mapa la moneda destino y el valor la tasa de cambio. Ejemplo:



8.12. (Opcional) Toma el proyecto del ejercicio 8.1 y añádele una imagen para cada empleado. Deberás seguir los pasos indicados en los apuntes y probarlo con Postman.

Tema 9: Seguridad y Control de Acceso

9.1. Crea un proyecto a partir del 7.1 *(CRUD de Empleado con Spring MVC sobre base de datos H2)* y configura la seguridad siguiendo los pasos mostrados en los apuntes, creando en el bean *UserDetailsService* dos usuarios, con nombre '*user*' y '*admin*', ambos con contraseña *1234* y con roles '*USER*' y '*ADMIN*' respectivamente. La vista de empleados deberá estar accesible a cualquier visitante, el usuario '*user*' podrá crear nuevos empleados, y el usuario '*admin*' podrá además editar y borrar empleados.

- Debes crear la clase *SecurityConfig* tal y como se muestra en los apuntes.
- La vista general de empleados contendrá un enlace a la página de *login* y otro a la página de */logout.*
- En caso de un acceso a una página a la que no se tiene permiso, se mostrará una página con un mensaje informando de esa situación.

9.2. Crea un proyecto a partir del 7.11 *(Cuentas corrientes y movimientos con Spring MVC sobre H2)* y configura la seguridad de la siguiente forma:

- Crea una enumeración que contenga los tres roles que tendremos en el sistema: administrador de la aplicación, titular de cuentas y resto de usuarios identificados.
- Crear una entidad "Usuario" (con atributos: id (autogenerado), nombre (sin repetidos), contraseña (como mínimo 4 caracteres) y rol. Crea un CRUD para esta entidad con repositorio, servicio, controlador y vistas. También deberás crear la implementación de la interfaz UserDetailService.
- En la vista de cuentas crea un enlace para la gestión de usuarios, botón de login y logout.
- Los permisos de cada rol son los siguientes:
  - Administrador: tendrá acceso a toda la aplicación (solo ellos pueden acceder al CRUD de Usuarios). Hará falta tener un primer usuario de tipo administrador creado por defecto para poder acceder a ese CRUD.
  - Titular: tiene acceso completo al CRUD de cuentas y movimientos, pero no al de usuarios.
  - Usuario: Tiene acceso completo al CRUD de movimientos, pero no al de cuentas, solo puede ver las cuentas, pero no crear ni editar ni eliminarlas.
  - Visitantes: (usuarios no identificados) solo tienen permiso para ver cuentas y movimientos.

9.3. Crea un proyecto a partir del anterior incorporando las siguientes modificaciones:

- Crea páginas de login y logout personalizadas.
- En la vista con la lista de cuentas, crea una capa que muestre el botón de login si no hay ningún usuario conectado o bien que muestre el nombre de usuario y el botón de logout en caso de que sí haya un usuario conectado.
- Modificar los permisos para que los usuarios que no sean titulares ni administradores solo puedan realizar ingresos y no retiradas; para ello deberás obtener el rol del usuario conectado en el servicio de movimientos, en el método de añadir movimiento.

9.4. Crea un proyecto a partir del 7.10 (*Tienda con productos, valoraciones y usuarios con Spring MVC y base de datos H2*) y configura la seguridad de la siguiente forma:

- Mover las rutas de las páginas genéricas (inicio, contacta, etc...) bajo una ruta /public.
- En el menú principal se añadirá un enlace para login y logout.
- Añadir a los usuarios el atributo contraseña, y rol (enumeración con valores: USER, MANAGER, ADMIN) y modificar el formulario de alta y edición de usuarios para incorporar los nuevos valores. Cada usuario tendrá un solo rol.
- Crear una clase de configuración de seguridad con los beans AuthenticationManager, PasswordEncoder y SecurityFilterChain deacuerdo a los siguientes permisos.
    - Las páginas bajo /public serán accesibles a cualquier visitante.
    - La vista general de productos y categorías también serán accesibles a cualquier visitante.
    - El rol USER permitirá añadir valoraciones de productos, pero no borrarlas (no hay edición de valorac.)
    - El rol MANAGER que podrá acceder al CRUD de productos, categorías y valoraciones de productos, pero no al CRUD de Usuarios.
    - El rol ADMIN al que tendrá a todas las operaciones de la aplicación. Es necesario crear un usuario inicial del este tipo, de lo contrario, no se podrán crear nuevos usuarios.
- La contraseña se almacenará encriptada en la base de datos, por lo que el servicio de usuarios tendrá inyectado el PasswordEncoder creado previamente y lo emplearemos para el encriptado al guardar un usuario.
- Crear la clase que implemente la interfaz UserDetailsService con el userRepository inyectado para obtener los datos del usuario que se desea loguear y cargar sus permisos.
- Si en la clase de configuración, en el bean SecurityFilterChain, hemos configurado el parámetro de página de error: `.exceptionHandling().accessDeniedPage("/accessError");` debemos crear el mapping y la vista correspondiente.
- Añadir en el menú de la aplicación (es un fragmento Thymeleaf) los enlaces a /login y /logout.

9.5. Crea un proyecto a partir del 9.4 mejorando ciertos aspectos:

- En el menú superior, cambiamos ligeramente los enlaces de login y logout: ahora se mostrará el enlace de login pero solo si no hay ningún usuario conectado; por el contrario, si hay un usuario conectado, se mostrará en ese menú el nombre del usuario, que será a su vez un menú desplegable con las opciones: editar perfil, cambiar contraseña y cerrar sesión *(piensa si sería adecuado crear un 'dto' para el cambio de contraseña)*.
- En el alta de nuevos usuarios, habría que controlar que el nombre de usuario sea único, que no lo tenga ningún otro usuario en el sistema. Y algo similar en la modificación de usuario (si se modifica el nombre, que el nuevo sea único).
- El rol USER permitirá añadir valoraciones de productos, pero solo borrar las realizadas por él mismo. Al añadir una valoración de producto, ya no será necesario introducir el usuario que realiza la valoración, ya que será el usuario que esté conectado.
- Añade la opción "recuérdame" basado en cookies cuando un usuario se autentifica.
- Añade al menú la opción de autoregistro (será con rol: USER). Los roles superiores serán asignados a posteriori solo por administradores.
- Y si no lo has hecho aún, crea bajo /templates/error páginas de error personalizadas: 404.html, 403.html y 500.html.

9.6. Crea un proyecto a partir del 8.4 *(CRUD de Empleado con API Rest y gestión de excepciones)* incorporando control de acceso con token JWT tal y como se muestra en el ejemplo entregado por el profesor. Tendrá las siguientes características:

- Habrá dos tipos de usuarios: USER y ADMIN.
- Si accede a la API un usuario no registrado solo podrán consultar la lista de empleados y acceder a las rutas de login y registro.
- Los usuarios de tipo USER podrán dar de alta empleados, pero solo podrán borrar y modificar empleados que hayan sido creados por ellos mismos. Puedes añadir una nueva excepción *"NoPermitidoException"* para aquellos que intenten realizar una edición/borrado sobre un empleado no creado por ellos *(será por tanto necesario modificar la clase Empleado para añadirle una relación con el usuario que lo ha creado)*.
- Los usuarios de tipo ADMIN tiene permisos completos sobre la aplicación.

Tema 10: Pase a producción y testing

10.1 Toma el ejercicio 7.1 y pasa la base de datos H2 a una base de datos MySQL.

- Instala MySQL y MySQL Workbench.
- Hay que crear previamente la base de datos vacía desde MySQL Workbench.

10.2 Toma el ejercicio anterior y haz, mediante perfiles de configuración, que en desarrollo emplee H2 y en producción MySQL, todo ello mediante diferentes archivos de propiedades, por ejemplo: *application-dev.properties, application-prod.properties.* El perfil a utilizar en cada momento se configurará en el archivo *application.properties* general.

10.3. Toma el ejercicio 8.1 e incorpora test unitarios para los métodos del controlador y servicio.

10.4. Incorpora *actuator* al ejercicio anterior, activa todos los endpoints, configura el endpoint *health* para que muestre más información y haz capturas de la información mostrada por algunos de ellos.

10.5. Realiza las siguientes prácticas en Docker

a) Toma el ejercicio 6.1 e instálalo en un contenedor.

b) Toma el ejercicio 7.6 e instálalo en un contenedor. Debes crear previamente un volumen.

c) Toma el ejercicio 10.1 e instálalo en un contenedor. Debes crear un contenedor adicional con MySQL.

10.6. Toma el proyecto de la tienda de temas anteriores (ejercicio 9.5) y modifícalo para incorporar todo lo visto en los ejercicios previos de este tema:

- Pasar la base de datos de H2 a MySQL en un servidor externo (Docker)
- Crea perfiles de ejecución en test y producción.
- Crea test unitarios para controladores y servicios.
- Incorpora *actuator* con todos los endpoints activos y que solo los administradores tengan acceso a ellos.

Anexo: Errores frecuentes

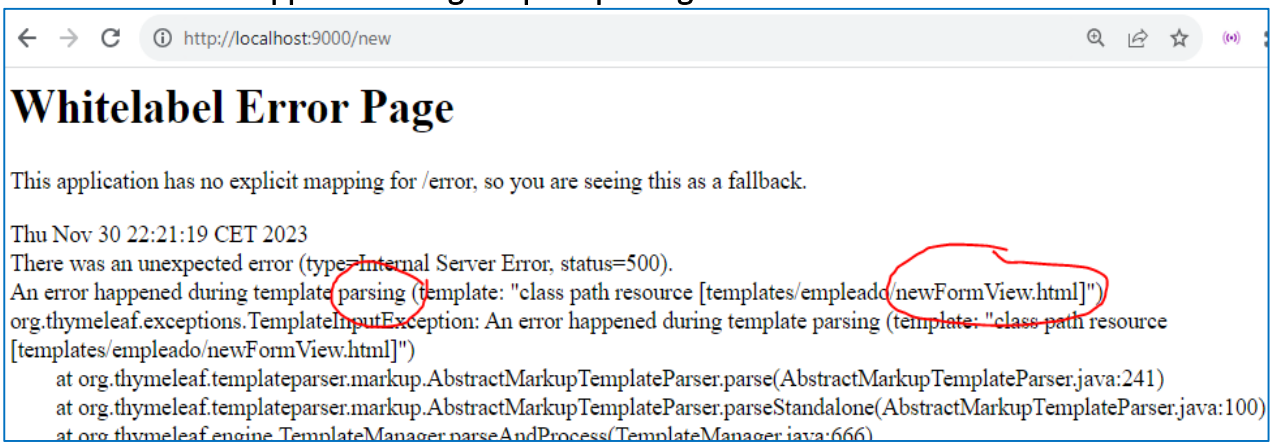
1. No informa de ningún error, pero no muestra los resultados en la vista.

Generalmente es un error de sintaxis: no coincide la variable pasada como parámetro en *model.addAttribute* con la variable reflejada en la vista:

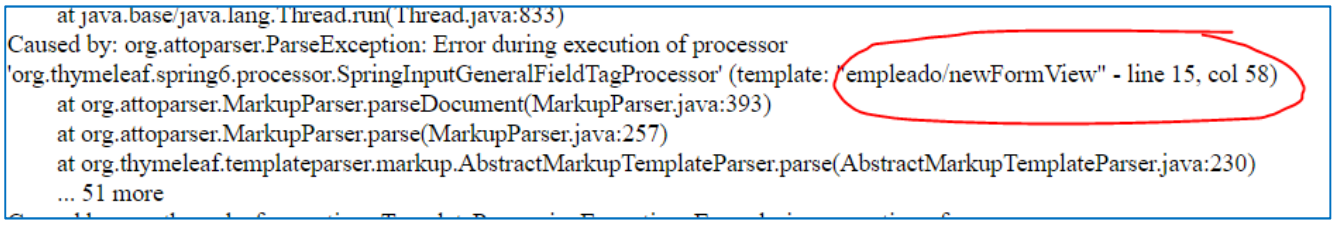
Vista: `<tr th:each="empleado : ${listaEmpledos}">`

Controlador: `model.addAttribute("listaEmpleados", empleadoService.obtenerTodos());`

2. Error 500: An error happened during template parsing

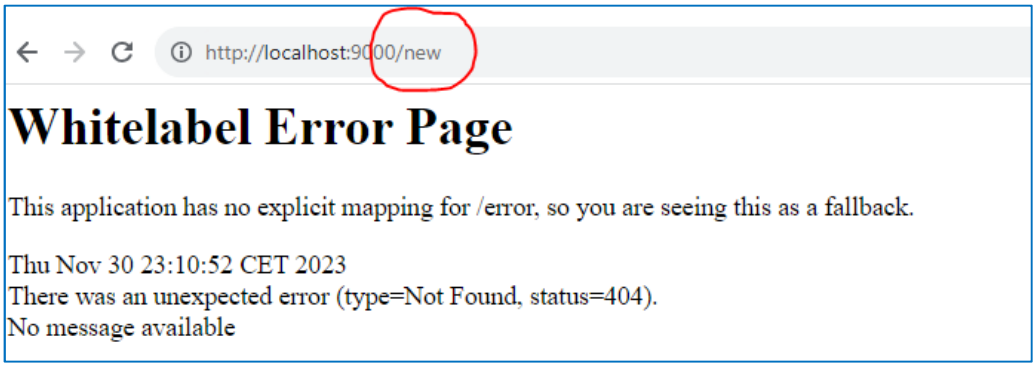


Error similar al anterior, no coincide la variable pasada como parámetro en *model.addAttribute* con la variable reflejada en la vista. Este mensaje de error, más abajo, informa de la línea en la que se produce el error.



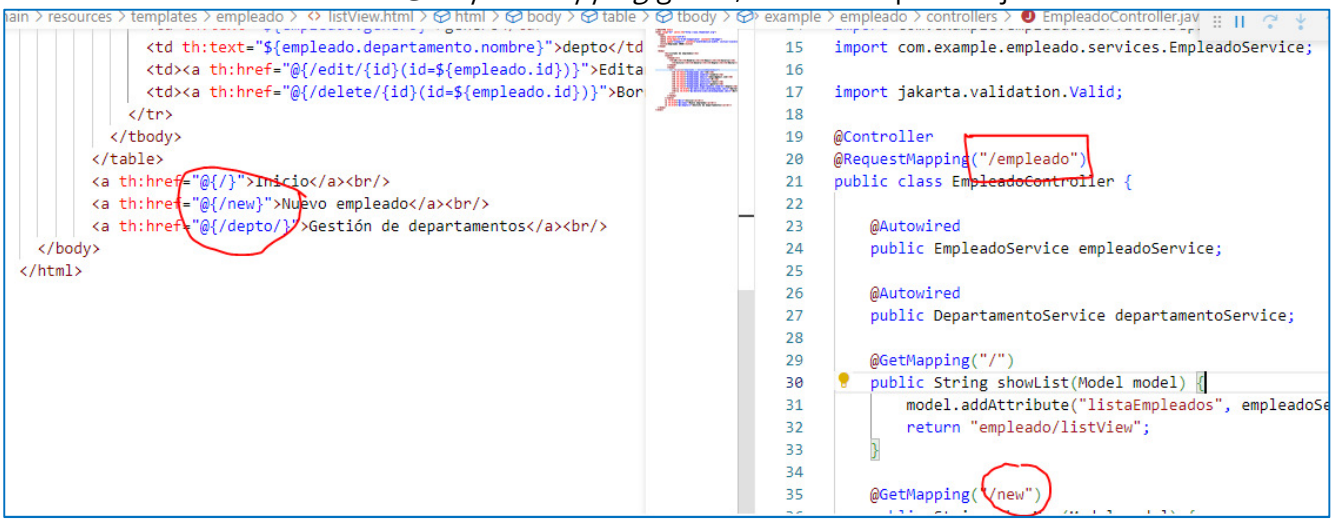
Es típico en formularios: erratas o bien que el objeto que se pasa al formulario no tiene getters y setters estándar sobre sus atributos.

3. Error 404: Not found



La URL solicitada (el mapping) no es tratada por ningún controlador. Esta URL puede ser escrita directamente por el usuario en la barra de direcciones del navegador o proceder de un enlace o botón de una vista. Puede ser debido a varias causas:

- Errata en la vista o en la anotación Mapping del controlador y estas no coinciden, por ejemplo, por mayúsculas/minúsculas.
- El controlador tiene un *@RequestMapping* global, este tiene que reflejarse en las URLs.



El error mostrado es debido a que la URL del enlace *"Nuevo empleado"* es `/new` y el mapping tratado por el controlador es `/empleado/new`.

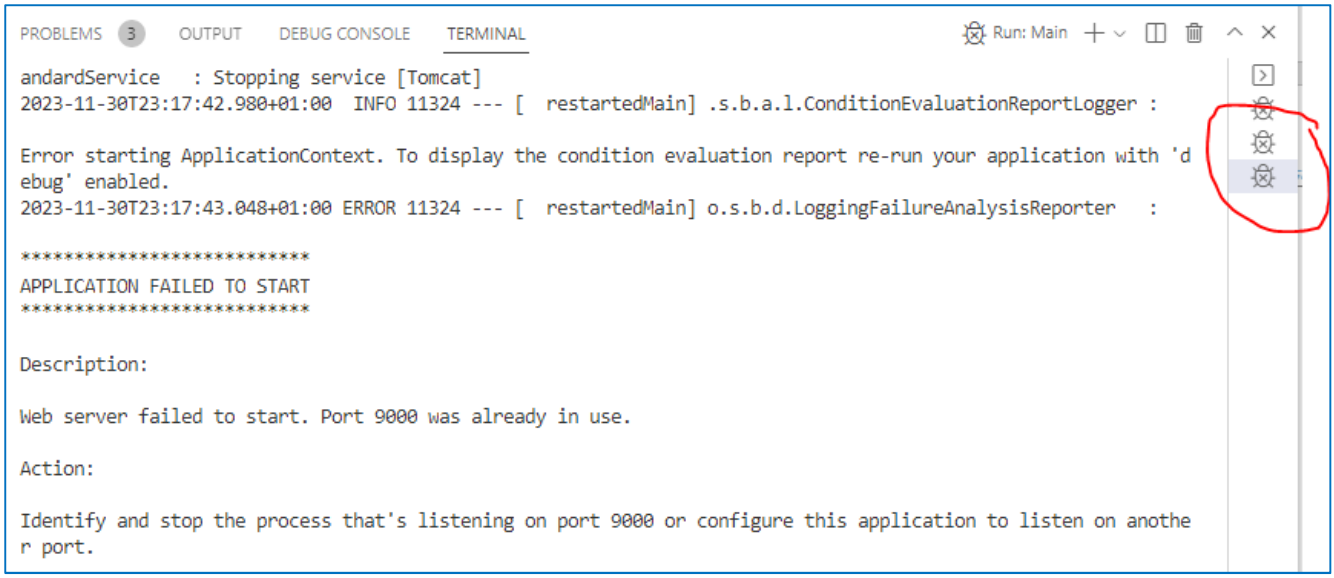
- El controlador no está situado en la carpeta de la clase "Main" de la aplicación o bien en una subcarpeta. Con SpringBoot todos los archivos Java deben cumplir este requisito: estar en la carpeta de la clase "Main" o bien en subcarpetas debajo de esta. Entendemos por clase "Main", la que está anotada con *@SpringBootApplicacion* y tiene el método main. Es en la clase que nos situamos para ejecutar la aplicación.

4. Required a bean of type... not found

```
***************************
APPLICATION FAILED TO START
***************************

Description:

Field departamentoService in com.example.empleado.controllers.DepartamentoController requ
ired a bean of type 'com.example.empleado.services.DepartamentoService' that could not be
 found.

The injection point has the following annotations:
        - @org.springframework.beans.factory.annotation.Autowired(required=true)


Action:

Consider defining a bean of type 'com.example.empleado.services.DepartamentoService' in y
our configuration.
```

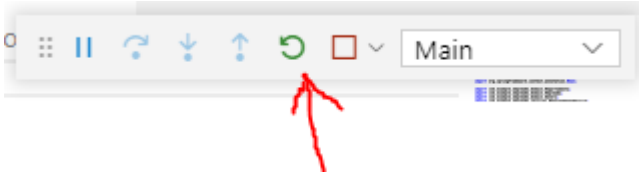
Falta anotación @Service, @Controller, etc en alguna clase de la aplicación.

5. Web server failed to start. Port ..... was already in use



Este error se produce cuando tenemos ejecutándose una aplicación y ejecutamos otra aplicación o bien volvemos a solicitar la ejecución de la misma de nuevo. Para solucionarlo hay que hacer un "Kill Terminal" de las sesiones previas para liberar el puerto, mediante los botones de la derecha en la consola.

Si estamos ejecutando una aplicación, hacemos una modificación, y queremos relanzarla, no ejecutaremos "Run", sino "ReStart":



6. Error en la primera línea de todos los archivos .java de la aplicación

Aparece un mensaje de error en la primera línea de todos los archivos java de la aplicación, la que indica el "package" en donde está ubicado el archivo.

Este error suele ser debido a que tenemos abierto en VSC no la carpeta del proyecto sino su carpeta "padre"; lo ideal con VSC es abrir la carpeta del proyecto, no carpetas por encima de esta.

7. Comportamientos extraños, no ejecuta el código modificado

En algunas ocasiones el IDE no compila las nuevas modificaciones y realiza un comportamiento extraño. En esos casos, suele ser una buena opción limpiar su caché. Para ello, desde el menú: *View > Command Palette > Clean Java Language Server WorkSpace*.

8. Could not determine recommended JdbcType for...

```
org.springframework.beans.factory.BeanCreationException: Error creating bean with name 'entityManagerFactory'
]: Could not determine recommended JdbcType for `com.example.empleado.domain.Departamento`
        at org.springframework.beans.factory.support.AbstractAutowireCapableBeanFactory.initializeBean(Abstra
        at org.springframework.beans.factory.support.AbstractAutowireCapableBeanFactory.doCreateBean(Abstract
        at org.springframework.beans.factory.support.AbstractAutowireCapableBeanFactory.createBean(AbstractAu
```

Falta anotación de relación entre entidades: @ManyToOne, @OneToOne, etc.