

Índice

Índice

Spring MVC

Controladores

- Consideraciones

Paso de información controlador-vista

Práctica 1. Controladores.

Introducción a Thymeleaf

- Mostrar información del modelo

- Condicionales IF y UNLESS

 - If

 - Unless

- Switch

- Bucle FOR

 - Ejemplo con String[] en una lista no numerada

 - Ejemplo con un ArrayList de objetos en una tabla HTML

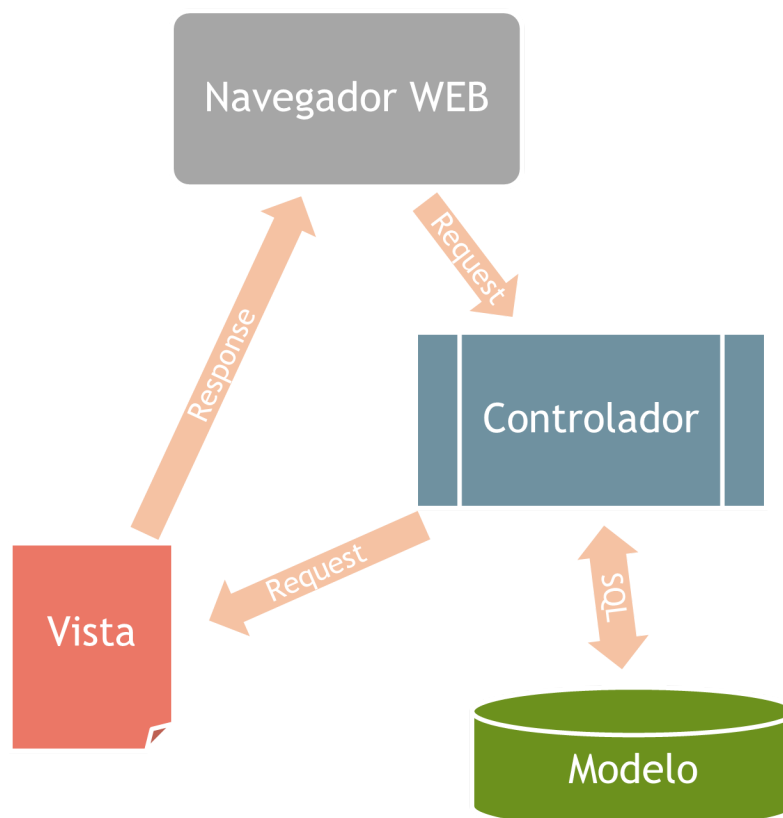
- Consideraciones

Práctica 2. Thymeleaf

Spring MVC

El framework Spring integra el patrón MVC (Modelo-Vista-Controlador). Expliquemos un poco en qué se basa el patrón MVC y después pasaremos a ver como se aplica dentro de Spring.

MVC consiste en separar el modelo de datos (base de datos), la vista que los representa (html, jsp, etc.), y la lógica de la aplicación (controladores).



Los dispositivos envían peticiones al controlador, que es el encargado de satisfacer los requerimientos del cliente, recuperando los valores del modelo de datos y enviándoselo a la vista, que será la encargada de representar la forma en que los datos se visualizarán. Esas vistas se envían a un navegador web que será el encargado de mostrarlo al cliente.

Cada capa será independiente del resto, por lo que no podremos, por ejemplo, hacer conexiones con la base de datos desde un archivo *jsp* o *thymeleaf*. La forma correcta sería hacer la lógica de la conexión desde un controlador (*servlet*), y enviar los datos necesarios por el *request* a la vista.

Controladores

El controlador es una evolución de los “antiguos” servlets. Gestiona peticiones (usualmente acciones del usuario) e invoca consultas al 'modelo' cuando se hace alguna solicitud sobre la información (por ejemplo, consultar un registro en una base de datos). También puede enviar objetos a su 'vista' asociada, por lo que podemos decir que el controlador hace de mediador entre el modelo de datos y las vistas.

Se pueden crear tantos controladores como sean necesarios, para organizar mejor el código. Por ejemplo, supongamos que tenemos una aplicación que gestiona pedidos, y tiene clientes y proveedores. Pues nuestra aplicación tendría un `HomeController` (equivalente al `FrontServlet`), `PedidosController`, `ClientesController` y `ProveedoresController`. En cada controlador ubicaremos los métodos para controlar cada parte de la aplicación. Realmente se podría hacer todo en un gigantesco Controlador, pero ya puedes imaginar que sería más complicado escalar y mantener esa aplicación.

En el **HolaMundo** anterior, ya deberíamos tener un `HomeController.java` creado, pero para entenderlo mejor, podemos borrar el archivo y empezamos de 0, explicando ahora si, paso a paso.

Para crear un nuevo controlador, seguiremos los siguientes pasos:

1. Creamos una clase normal en un paquete adecuado. Normalmente en un paquete llamado `controllers`.

```
1 package com.ejemplo.holamundo.controllers;
2
3 public class HomeController {
4
5 }
```

2. Para convertir esa clase en un **Controller**, solo tendremos que usar la anotación `@Controller` sobre la clase y hacer la importación. Ya tenemos nuestro controlador creado. No hay más.

```
1 package com.ejemplo.holamundo.controllers;
2 import org.springframework.stereotype.Controller;
3
4 @Controller
5 public class HomeController {
6
7 }
```

3. Con el siguiente código, hacemos que cualquier petición a la url indicada en la anotación `@RequestMapping("url")`, Spring realizará una llamada al método indicado.

```

1 | @Controller
2 | public class HomeController {
3 |     @RequestMapping("/")
4 |     public String home() {
5 |         return "home";
6 |     }
7 | }

```

4. En nuestra aplicación, cada vez que vayamos a la url `/` nos llevará a la vista `home.html`.
5. La vista `home.html`, es tan sólo un archivo HTML convencional en el que al indicarle el atributo `xmlns:th="http://www.thymeleaf.org"` le estamos proporcionando una serie nueva de atributos a las etiquetas de HTML que dotarán a las vistas de la capacidad de ser páginas dinámicas de servidor, que se procesarán en tiempo de ejecución (como las antiguas JSP). En nuestro ejemplo, no usamos nada especial de Thymeleaf, por ahora.

Consideraciones

👉 Con la anotación `@RequestMapping("url")` le indicamos que cuando la app se dirija a esa url, se debe ejecutar el método `home()`. El nombre del método puede llamarse como quieras, pero debería ser descriptivo.

👉 Con `@RequestMapping("url")`, recibiremos las peticiones que vengan tanto por `GET` como por `POST`. También podemos diferenciarlas si lo necesitamos con `@GetMapping()` y `@PostMapping()`.

👉 Los métodos siempre seguirán la misma estructura, es decir, puede llamarse como quieras, y devolverá un `String`. Ese `String` será el nombre de la vista a la que va a dirigirse. No hace falta indicar el prefijo `src/main/resources/templates`. No hace falta indicar el sufijo `.html`.

👉 No necesitamos `RequestDispatcher`, `forward`, ni crear un servlet individual con su `doGet` y/o `doPost` por cada url a controlar. Creamos un controlador y añadimos tantos métodos con su correspondiente `RequestMapping` como queramos.

👉 Hay muchas otras firmas de métodos que Spring también reconocería, pero esa es la más extendida y simple. Siempre puedes mirar la [documentación oficial](#) y flipar 🤖.

👉 Con el uso de anotaciones (presentes desde la versión 2.5 de Spring), nos hemos evitado tener que declarar cada Controlador y cada vista en laberínticos archivos xml de configuración, y muchas más ventajas que veremos más adelante, como el paso de parámetros por anotaciones, recuperar e insertar objetos en la sesión, sistemas de seguridad implícitos y en definitiva **haciendo más fácil las tareas cotidianas y repetitivas** de una aplicación web.

Paso de información controlador-vista

Vamos a enviar alguna información desde el controlador a la vista. Es decir, desde el código de la capa de negocio en `HomeController.java` hasta la capa de la vista en `home.html`.

Recordemos como se hace en JavaEE:

```

1 public class HomeServlet extends HttpServlet {
2     public void doPost(HttpServletRequest request, HttpServletResponse
response)
3         throws ServletException, IOException {
4
5         //Adjuntamos mensaje en el request
6         String saludo = "¡¡Propicios días!! Mensaje desde el controlador";
7         request.setAttribute("mensaje", saludo);
8
9         //Enviamos el request a home.html
10        RequestDispatcher rd = request.getRequestDispatcher("home.html");
11        rd.forward(request, response);
12    }
13 }

```

Necesitábamos el objeto `request` para adjuntarle lo que quisiéramos con el método `request.setAttribute()`, para después recuperarlo desde JSP con `request.getAttribute()`, o usando la librería JSTL. Además de toda la ceremonia que era tratar con el objeto `request` de un sitio para otro.

Viendo el código del controlador anterior `HomeController`, ¿Dónde está el `request`? ¿Si no tengo el `request`, como adjuntamos objetos a las vistas? ¿O como sacamos el `session`? **La solución inyectar los objetos** que necesitemos en los métodos de los controladores.

Ahora veremos como se hace en Spring:

```

1 @RequestMapping("/")
2 public String home(Model model) {
3     String mensaje = "¡¡ Propicios días!! Mira mama, en dos líneas!!";
4     model.addAttribute("saludo", mensaje);
5     return "home";
6 }
7

```

- Lo único que hacemos es inyectar (poner un objeto de la clase Model) en los parámetros del método y ya lo tenemos disponible para usarlo.
- El objeto de la clase Model representa el modelo de datos, y podemos adjuntarlo cualquier otro objeto, usando su método:
 - `addAttribute(string, object)` -> Recibe un String y un Object. El String indica el nombre con el que vamos a guardar el Object dentro del modelo. Igual que el `setAttribute` de `Request`, `Session` y `Application`.
 - No contiene un método tipo `removeAttribute()`, por lo que si queremos eliminar un objeto del model (algo raro), siempre podemos hacer `addAttribute("mensaje", null)`.
- Otros objetos que podemos inyectar son el `Request`, `Response`, `Session` y con inyectarlos de la misma forma que el model, podríamos usarlos en el método. Una mala práctica sería inyectar objetos que no se vayan a usar, por ejemplo, si no vamos a guardar objetos en el modelo, no lo inyectamos en el método.
- Podemos inyectar cualquier objeto que necesitemos en los métodos que los necesitemos.

```

1  @RequestMapping("/")
2  public String home(Model model, HttpSession session) {
3      GregorianCalendar calendario = new GregorianCalendar();
4      String ahora = calendario.getTime().toString();
5
6      model.addAttribute("saludo", "¡¡ Propicios días !!");
7      model.addAttribute("serverTime", ahora);
8      model.addAttribute("sessionId", session.getId());
9
10     return "home";
11 }

```

- Como vemos en el ejemplo anterior, inyectamos el `Model` y el `HttpSession`, y obtenemos la id de la sesión con `session.getId()` y la añadimos como atributo al `model`. Y no nos olvidamos de enviar un saludo, la educación por delante 😊.

Ya hemos enviado los atributos del controlador a la vista, ahora nos quedaría como recuperar esos atributos y mostrarlos en la página `home.html`.

En la vista podemos tener el siguiente código:

```

1  <body>
2      <h1>Hola mundo!!!</h1>
3      <h3 th:text="${saludo}"></h3>
4      <ul>
5          <li>
6              <span>Hora del servidor:</span>
7              <span th:text="${serverTime}"></span>
8          </li>
9          <li>
10             <span>ID de sesión:</span>
11             <span th:text="${sessionId}"></span>
12         </li>
13     </ul>
14 </body>

```

Con los atributos `th:*` podemos acceder a los atributos que hay en el `model` y mostrarlos directamente en la vista en los elementos HTML que queramos.



Práctica 1. Controladores.

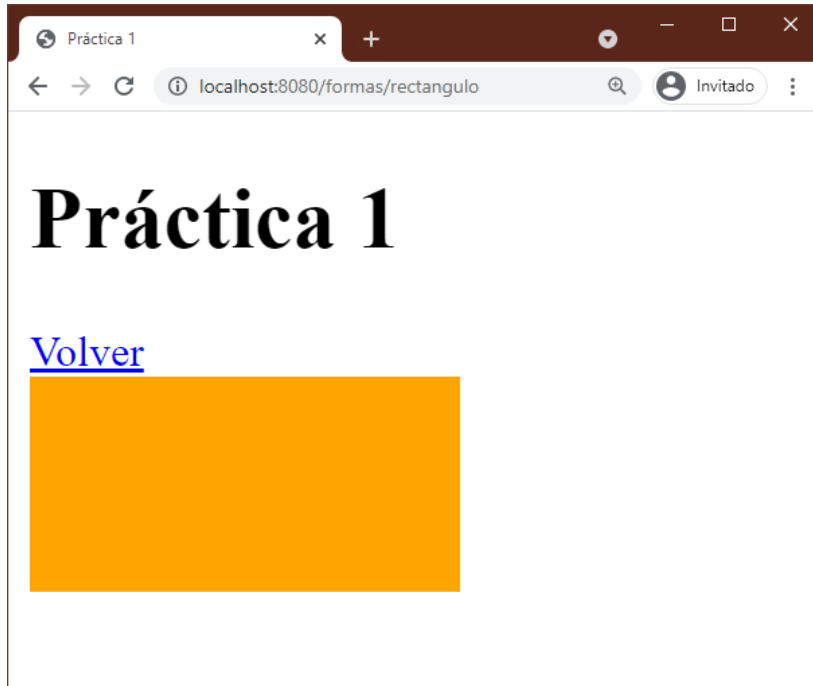
Crear un proyecto nuevo Spring Boot, con las dependencias de `Spring MVC` y `Thymeleaf`.

- Tendrá un controlador llamado `HomeController` que al llegar una petición a la url raíz (`/`) llevará a una vista llamada `index.html`.
- La vista `index.html` tendrá el siguiente contenido:
-



- Cada enlace llevará a dos vistas, `cuadrado.html` y `rectangulo.html`. Ambas estarán en una carpeta llamada `/pages` en la carpeta donde van las vistas.
- Para dirigir desde un controlador a las vistas que queremos, vamos a crear un **nuevo controlador** llamado `FormasController`. Podríamos hacerlo en el mismo `HomeController`, pero así hacemos nuestro programa más escalable.
- Las rutas para ir a las vistas serán:
 - Para ir a `cuadrado.html` usaremos la url `/formas/cuadrado`
 - Para ir a `rectangulo.html` usaremos la url `/formas/rectangulo`
 - Se pueden crear *mappings* parciales asignando una parte de la ruta al controlador, y otra parte al método. `/formas`, `/cuadrado` y `/rectangulo`, en lugar de `/formas/cuadrado` y `/formas/rectangulo`. Sería una buena idea hacerlo aquí.

- 🔗 Prueba primero las rutas a mano en el navegador, y cuando funcione usa esas urls para los enlaces.
- 🔗 Experimenta con `@RequestMapping` y `@GetMapping`.
- Los enlaces de `index.html` usarán las rutas que hemos definido en los controladores. Prueba a usar la ruta de una vista directamente.
- En las vistas `rectangulo.html` y `cuadrado.html`, tendrán lo que prometen. Un cuadrado y un rectángulo. Y un enlace que nos vuelva a `index.html`. Las figuras las puedes hacer con un `<div>` y dándole directamente un `width`, `height` y un `background-color`. Puedes usar estilos con el atributo `style=""` o con la etiqueta `<style>` **desde el mismo HTML**.



- 🔗 Para usar un archivo externo de CSS, ¿en qué ruta deberemos colocarlos? ¿Cómo los referenciamos?

Introducción a Thymeleaf

Thymeleaf es una biblioteca Java que implementa un motor de plantillas HTML5 que puede ser usado en entornos web. Se acopla perfectamente para trabajar en la capa vista del MVC. Proporciona un módulo opcional para la integración con Spring MVC, por lo que se usa para reemplazar completamente a los archivos JSP en las aplicaciones web.

El objetivo principal de Thymeleaf es permitir la creación de plantillas de una manera elegante y un código formateado, sin usar etiquetas nuevas de HTML, si no extendiendo sus capacidades con nuevos atributos. Por esta razón, obtendremos un archivo HTML fácil de entender sin rastro de código java, como podía pasar en las JSP.

También ofrece la característica llamada **plantillas naturales**, que se basa en que un navegador pueda abrir la plantilla de forma estática (sin procesar) y se pueda mostrar sin problemas. Esto permitirá a los equipos de diseño y desarrollo trabajar en el mismo archivo de plantilla y reducirá el esfuerzo requerido para transformar un prototipo estático en un archivo de plantilla funcional.

Podemos encontrar la documentación oficial en <https://www.thymeleaf.org/documentation.html>.

A continuación vamos a ver ejemplos de las estructuras básicas de cualquier lenguaje de programación que se pueden usar en Thymeleaf.

Mostrar información del modelo

Para mostrar cualquier objeto enviado desde el controlador a la vista HTML, usaremos el atributo `th:text` en el elemento HTML que queramos, y así sustituir el contenido HTML por el valor del atributo:

```
1 //Desde el controlador java...
2 String usuario="salva@formador.es";
3 model.addAttribute("nombre", usuario);
```

```
1 <!-- Desde HTML -->
2 <p th:text="{nombre}">El contenido será eliminado</p>
```

Como valor del atributo `th:text` se usa *Expression Language* (EL). En resumen, se usa la sintaxis `{ }` y se evalúa la expresión de su interior. Puede ser el contenido de una variable u otra expresión más compleja. Por ejemplo `{nombre.toLowerCase() + '!!!' }`

El texto de `El contenido del párrafo será eliminado`, será sustituido en tiempo de compilación por el resultado de la expresión `nombre`, por lo que la plantilla una vez compilada, quedará de la siguiente forma:

```
1 <p>salva@formador.es</p>
```

También podemos usar la siguiente sintaxis para mostrar expresiones dentro del contenido de las etiquetas. Los siguientes párrafos obtendrían el mismo resultado en pantalla. En muchos casos donde se intercalan varios textos fijos y expresiones, mejoraría la legibilidad. Igual que los *string templates* de JavaScript.

```
1 <p>Bienvenido, [{usuario.nombre}]. Tu rol es [{usuario.rol}].</p>
2 <p>Bienvenido, <span th:text="{usuario.nombre}"></span>. Tu rol es <span
  th:text="{usuario.rol}"></span>.</p>
```

Condicionales IF y UNLESS

If

Se pueden hacer estructuras condicionales con el atributo `th:if`.

```
1 <div th:if="{user.esAdmin()}">Panel de administrador</div>
```

Al compilar la plantilla, INCLUIRÁ el elemento sólo si la expresión indicada es `true`. En caso contrario no se incluirá elemento

en el HTML de respuesta. La vista quedaría así en caso de ser la expresión verdadera.

```
1 <div>Panel de administrador</div>
```


Unless

Con el atributo `th:unless` hacemos justamente lo contrario. Sería como el bloque `else` de cualquier lenguaje de programación.

```
1 <a th:unless="${edad < 18}" href='www.paginasdemayores.com'>Página Secretas</a>
```

Si el usuario es menor de edad NO se mostraría el enlace.

Switch

También posee estructuras de tipo **switch-case**. Así nos evitaríamos en algunos casos varias sentencias `th:if` y quedaría un código más limpio y escalable. Se usaría la siguiente sintaxis:

```
1 <div th:switch="${user.rol}">
2   <p th:case="'admin'">Usuario es administrador</p>
3   <p th:case="${roles.manager}">Usuario es manager</p>
4   <p th:case="${roles.editor}">Usuario es editor</p>
5   <p th:case="*">El usuario es otra cosa</p>
6 </div>
```

Bucle FOR

También podemos recorrer una colección de elementos con una estructura **FOR**. Es muy parecido a un `for each`. Repite el elemento HTML tantas veces como elementos haya en la colección.

Desde el controlador que deseemos generamos la colección que queramos y la adjuntamos al `model`.

Ejemplo con `String[]` en una lista no numerada

```
1 @GetMapping("/test/for-array")
2 public String pruebaArray(Model model) {
3     String listaCompra[] = {"Pepinos", "Tomates", "Patatas", "Pimientos"};
4     model.addAttribute("lista", listaCompra);
5     return "lista-compra";
6 }
```

Y desde HTML la mostramos así:

```
1 ...
2 <body>
3     <h3>Lista de la compra</h3>
4     <ul>
5         <li th:each="articulo : ${lista}">
6             <span th:text="${articulo}"></span>
7         </li>
8     </ul>
9 </body>
```

Una vez compilada, obtendríamos el siguiente código fuente en el cliente:

```

1  ...
2  <body>
3      <h3>Lista de la compra</h3>
4      <ul>
5          <li>
6              <span>Pepinos</span>
7          </li>
8          <li>
9              <span>Tomates</span>
10         </li>
11         <li>
12             <span>Patatas</span>
13         </li>
14         <li>
15             <span>Pimientos</span>
16         </li>
17     </ul>
18 </body>

```

Ejemplo con un ArrayList de objetos en una tabla HTML

Ahora en lugar de un array simple de Strings, vamos a crear un `ArrayList<Articulo>` y mostraremos esos datos en una tabla HTML

En el controlador:

```

1  @GetMapping("/test/for-arraylist")
2  public String pruebaArrayList(Model model) {
3      List<Articulo> listaArticulos = new ArrayList<Articulo>();
4      listaArticulos.add(new Articulo(1, "Agua Chumicero 1.5L", 0.70));
5      listaArticulos.add(new Articulo(3, "Refresco con gas sabor cola",
6      1.35));
7      listaArticulos.add(new Articulo(1, "Campero Pollo y Bacon XXL", 6));
8      listaArticulos.add(new Articulo(1, "Pizza Barbacoa", 7));
9      listaArticulos.add(new Articulo(1, "Ración Patatas fritas", 3.2));
10
11      model.addAttribute("lista", listaArticulos);
12
13      return "home";
14  }

```

En la vista:

```

1  <h3>Ticket</h3>
2  <table>
3      <tr>
4          <th>Cantidad</th>
5          <th>Descripción</th>
6          <th>Precio</th>
7          <th>Total</th>
8      </tr>
9      <tr th:each="articulo : ${lista}">
10         <td th:text="${articulo.cantidad}"></td>
11         <td th:text="${articulo.descripcion}"></td>
12         <td th:text="${articulo.precio}"></td>
13         <td th:text="${articulo.cantidad * articulo.precio + ' €'}"></td>

```

```
14     </tr>
15 </table>
```

Generaría el siguiente HTML final:

```
1  <h3>Ticket</h3>
2  <table>
3    <tr>
4      <th>Cantidad</th>
5      <th>Descripción</th>
6      <th>Precio</th>
7      <th>Total</th>
8    </tr>
9    <tr>
10     <td>1</td>
11     <td>Agua Chumicero 1.5L</td>
12     <td>0.7</td>
13     <th>0.7€</th>
14   </tr>
15   <tr>
16     <td>2</td>
17     <td>Refresco con gas sabor cola</td>
18     <td>1.35</td>
19     <th>2.7€</th>
20   </tr>
21   <tr>
22     <td>3</td>
23     <td>Campero Pollo y Bacon XXL</td>
24     <td>6.0</td>
25     <th>18.0€</th>
26   </tr>
27   <tr>
28     <td>4</td>
29     <td>Pizza Barbacoa</td>
30     <td>7.0</td>
31     <th>28.0€</th>
32   </tr>
33   <tr>
34     <td>5</td>
35     <td>Ración Patatas fritas</td>
36     <td>3.2</td>
37     <th>16.0€</th>
38   </tr>
39 </table>
```

Consideraciones

Hay que tener en cuenta que para que Thymeleaf funcione correctamente usando las Spring EL como `${artículo.nombre}`, el objeto `Artículo` deberá:

- Tener un constructor por defecto siempre. Aunque no sea estrictamente necesario ahora, es aconsejable que nos acostumbremos para cuando usemos beans, que si que será obligatorio para la persistencia con JPA.

- **Tener los *getters* y *setters* bien contruidos.** Los identificadores serán `get` o `set` + nombre del atributo con la primera letra en mayúsculas. Ejemplo: `getPrecio()` `setPrecio()`. Y si es el getter de un booleano se llamará `is` + nombre del atributo booleano con la primera en mayúsculas. Ejemplo: `isAdmin()`. No sería válido `getAdmin()`.
- Si tienen los getters y setters bien, al escribir `${articulo.precio}` estaremos haciendo una llamada al método `articulo.getPrecio()`. De ahí que tengan que estar de una forma concreta. Siempre se podrán llamar directamente con `${articulo.getPrecio()}`.
- Tener el método `.toString()` implementado. Si imprimimos directamente `${articulo}` hará una llamada a `articulo.toString()` por lo que debería existir.
- También desde Thymeleaf podemos llamar directamente a los métodos de las clases, aunque no sean *getters* ni *setters*:

```
1 <th th:text="${articulo.calcularPVP()}"></th>
```

Práctica 2. Thymeleaf

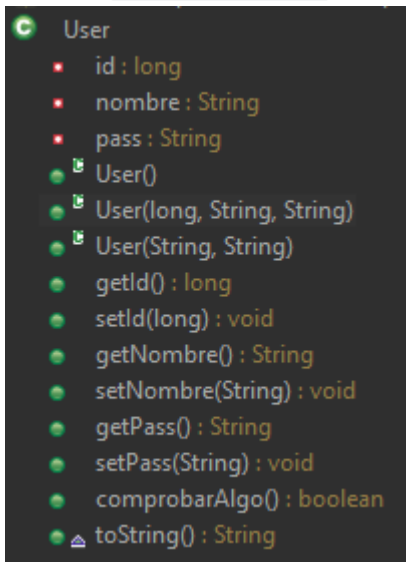
Hacer una aplicación Spring Boot MVC (Spring Starter Project) que tenga:

Tendrá las siguientes vistas:

- `home.html` -> Añádele algunos elementos HTML de atrezzo simulando una barra de navegación. Deberá tener un enlace que nos lleve a `listado.html` (nos llevará a un controlador que a su vez nos llevará a la vista, nunca de vista a vista).
- `listado.html` -> Se mostrará un listado en una tabla HTML.

Tendrá el siguiente POJO:

- `User.java`. En la siguiente imagen se detallan los atributos (rojo) y métodos (verde). El método `comprobarAlgo()` devuelve de forma aleatoria o un `true` o un `false`.





```
User
  id : long
  nombre : String
  pass : String
  User()
  User(long, String, String)
  User(String, String)
  getId() : long
  setId(long) : void
  getNombre() : String
  setNombre(String) : void
  getPass() : String
  setPass(String) : void
  comprobarAlgo() : boolean
  toString() : String
```

Tendrá los siguientes controladores:

- `HomeController` -> URL Mapeada `/`: Nos redirige inicialmente a `home.html` directamente.
- `ListadosController` -> URL Mapeada `/listados`: Creará un `ArrayList` de Objetos `User`. Los usuarios se crearán manualmente (5 ó 6). Se añaden también manualmente al `ArrayList` y se lo enviará a la vista `listado.html`.

✓ En la vista `listado.html` se mostrará en una tabla HTML, visualizando cada elemento del `ArrayList` (usuario) en una fila.

- ✓ En la última columna usar el resultado que nos devuelva el método `comprobarAlgo()`, para mostrar SI o NO (`true` o `false` respectivamente). Se puede usar un icono, imagen, emoji ( ) o simple texto, como quieras, pero no podrá salir true o false directamente.
- ✓ Mostraremos en algún sitio del listado, donde te parezca más oportuno, el número de elementos que tiene la tabla (número de usuarios que hay en la lista).
- ✓ Podemos añadirle un método a la clase `User`, que devuelva la contraseña oculta (sustituir cada carácter por un asterisco). Mostrar en el listado la contraseña oculta.
- ✓ Distribuir los objetos Java en paquetes. No mezclar los controladores con los pojos.