Sistemas Distribuidos e Internet

Sesión, Roles, Consultas, Búsqueda y Paginación

Sesión- 4

Curso 2017/ 2018

Contenido

1 Sesión

Al igual que en la mayor parte de tecnologías de desarrollo en el servidor disponemos del objeto **sesión**. Este objeto se utiliza para almacenar y recuperar datos correspondientes a la sesión de un usuario, estos datos permacen en el servidor hasta la expiración de la sesión.

Algunos usos comunes de la sesión son:

- Identificar al usuario autenticado (aunque en este caso lo estamos haciendo con spring security),
- Guardar datos temporales como los productos que metemos en el carrito en una tienda.

Vamos a modificar la aplicación para guardar en sesión una lista de las ultimas notas consultadas (ultimas notas vistas en detalles).

Accedemos a **MarkService** e inyectamos la sesión, objeto **HttpSession**.

```
@Service
public class MarksService {

	@Autowired
	private HttpSession httpSession;

	@Autowired
	private MarksRepository marksRepository;
```

Cada vez que visitamos los detalles de una nota **/getMark(Long id)** vamos a almacenar la información de la nota en sesión (también podríamos almacenar unicamene el identificador de la nota).

Los atributos del objeto **HttpSession** se consultan con la función **getAttribute(<clave del atributo>).**

Obtenemos el objeto con clave **consultedMarks**, como es la primera vez que obteneos la la lista de notas consutadas esta puede ser null, si es null tenemos que inicializarla.

Después agregamos la nota a la lista de notas consultadas y la volvemos a guardar en sesión con la función **setAttribute(<clave del atributo>, valor);**

```
public Mark getMark(Long id){
	Set<Mark> consultedList = (Set<Mark>) httpSession.getAttribute("consultedList");
	if ( consultedList == null ) {
		consultedList = new HashSet<Mark>();
	}
	Mark markObtained = marksRepository.findOne(id);
	consultedList.add(markObtained);
	httpSession.setAttribute("consultedList", consultedList);
	return markObtained;

}
```

Modificamos el controlador **MarksController**, inyectamos la sesión

```
@Controller
public class MarksController {

	@Autowired
	private HttpSession httpSession;
```

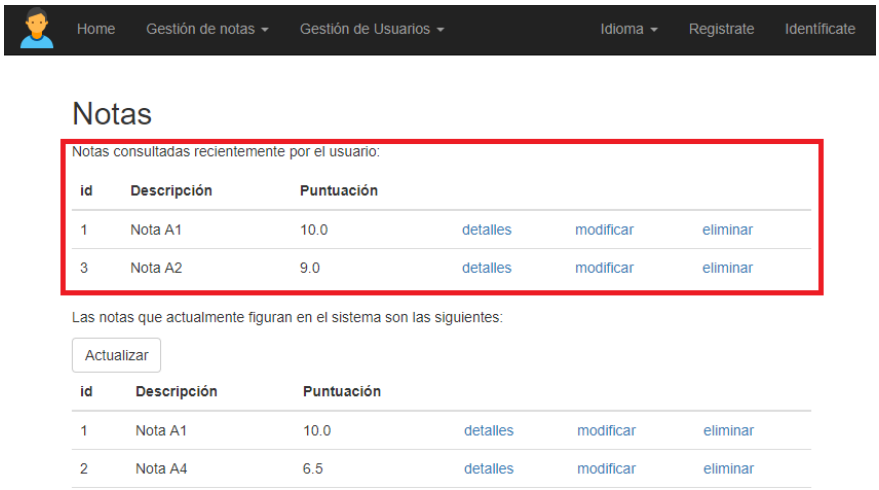
En la función que responde a **/mark/list** obtenemos el atributo de la sesión que contiene la lista de notas consultadas (debemos tener en cuenta que el atributo puede ser null), enviamos la lista a la vista bajo el nombre **consultedList.**

```
@RequestMapping("/mark/list")
public String getList(Model model){
	Set<Mark> consultedList= (Set<Mark>) httpSession.getAttribute("consultedList");
	if ( consultedList == null ) {
		consultedList = new HashSet<Mark>();
	}
	model.addAttribute("consultedList", consultedList);

	model.addAttribute("markList", marksService.getMarks() );
	return "mark/list";

}
```

En la vista **mark/list** creamos una nueva tabla al inicio del contenedor principal que mostrará las notas consultadas recientemente y que están almacenadas en **consultedList**.

```
<div class="container">
  <h2>Notas</h2>
  <p>Notas consultadas recientemente por el usuario:</p>
  <div class="table-responsive">
  <table class="table table-hover">
    <thead>
      <tr>
        <th class="col-md-1">id</th>
        <th>Descripción</th>
        <th>Puntuación</th>
                <th class="col-md-1"> </th>
                <th class="col-md-1"> </th>
                <th class="col-md-1"> </th>
      </tr>
    </thead>
        <tbody>
          <tr th:each="mark : ${consultedList}">
            <td th:text="${mark.id}"> 1</td>
            <td th:text="${mark.description}"> Ejercicio 1</td>
            <td th:text="${mark.score}">10</td>
            <td><a th:href="${'/mark/details/' + mark.id}">detalles</a></td>
            <td><a th:href="${'/mark/edit/' + mark.id}">modificar</a></td>
            <td><a th:href="${'/mark/delete/' + mark.id}">eliminar</a></td>
          </tr>
        </tbody>
  </table>
  </div>
  <p>Las notas que actualmente figuran en el sistema son las siguientes:</p>
    <button type="button" id="updateButton" class="btn btn-default">Actualizar</button>
  <script>
          $( "#updateButton" ).click(function() {
                  $("#tableMarks").load('/mark/list/update');
          });

  </script>
```

Sí nos identificamos con un usuario (99999990A contraseña: 123456) y entramos en los detalles de varias notas veremos las ultimas notas desde la vista **/mark/list**



Tambien podemos gestionar la sesión desde la propia plantilla, utilizando el objeto **session**.

${session.isEmpty()} // Comprobar si la sesión está vacia

${session.containsKey('consultedList ')} // Comprobar si la sesión contiene un atributo

${session.consultedList} // Acceder al atributo

1.1 Volver a la versión anterior

Importante: Una vez comprobada la funcionalidad del manejo de sesión y para simplificar la aplicación vamos a eliminar de la vista **/mark/list** la tabla de "notas consultadas recientemente por el usuario".

```
<div class="container">
  <h2>Notas</h2>
  <p>Notas consultadas recientemente por el usuario:</p>
  <div class="table-responsive">
  <table class="table table-hover">
    <thead>
      <tr>
        <th class="col-md-1">id</th>
        <th>Descripción</th>
        <th>Puntuación</th>
                <th class="col-md-1"></th>
                <th class="col-md-1"></th>
                <th class="col-md-1"></th>
      </tr>
    </thead>
        <tbody>
          <tr th:each="mark : ${consultedList}">
            <td th:text="${mark.id}">1</td>
            <td th:text="${mark.description}">Ejercicio 1</td>
            <td th:text="${mark.score}">10</td>
            <td><a th:href="${'/mark/details/' + mark.id}">detalles</a></td>
            <td><a th:href="${'/mark/edit/' + mark.id}">modificar</a></td>
            <td><a th:href="${'/mark/delete/' + mark.id}">eliminar</a></td>
          </tr>
        </tbody>
```

```
</table>
</div>

  <p>Las notas que actualmente figuran en el sistema son las siguientes:</p>
    <button type="button" id="updateButton" class="btn btn-default">Actualizar</button>
  <script>
          $( "#updateButton" ).click(function() {
                  $("#tableMarks").load('/mark/list/update');
          });

  </script>
```

Tambien accedemos al controlador **MarksController** y eliminamos el uso de la sesión.

```
@RequestMapping("/mark/list")
public String getList(Model model){
        Set<Mark> consultedList = (Set<Mark>) httpSession.getAttribute("consultedList");
        if ( consultedList == null ) {
                consultedList = new HashSet<Mark>();
        }
        model.addAttribute("consultedList", consultedList);

        model.addAttribute("markList", marksService.getMarks() );
        return "mark/list";

}
```

2 Roles

Vamos a permitir tres tipos de roles en la aplicación ROLE_STUDENT, ROLE_PROFESSOR, ROLE_ADMIN, dependiendo del role se permitirá acceder a unas funcionalidades o a otras.

- **/user/add** agregar usuarios con cualquier ROLE nos permitirá elegir el ROLE.
- **/login** identificar usuario (mismo funcionamiento que el actual)
- **/singup** registrarse (solo usuario con ROLE_STUDENT).

2.1 Actualización de entidades

Agregamos el atributo **role** a la entidad **User**

```
private String role;
```

Agregamos métodos get y ser para **role**

```
public String getRole() {
        return role;
}

public void setRole(String role) {
        this.role = role;

}
```

2.2 RolesService, servicio para la gestión de roles

Creamos la clase **RolesService** en el paquete **com.uniovi.services**, implementamos la función **getRoles()** que retorna la lista con los roles. *Vamos a incluir los roles en una lista, pero lo ideal sería incluir una entidad Role en bases de datos y relacionarla con la entidad user.*

Servicio gestión de roles **RolesService**, retorna la lista de los roles.

```
@Service
public class RolesService {
	String[] roles = {"ROLE_STUDENT","ROLE_PROFESSOR","ROLE_ADMIN"};

	public String[] getRoles() {
		return roles;
	}
}
```

2.3 InsertSampleDataService generación de datos de prueba

Modificamos nuestro servicio de pruebas **InsertSampleDataService** para incluir un role a los usuarios. Inyectamos el servicio **RolesService** para obtener el nombre de los roles y evitar confundirnos al escribir el String.

```
@Service
public class InsertSampleDataService {

	@Autowired
	private UsersService usersService;

	@Autowired
	private MarksService marksService;

	@Autowired
	private RolesService rolesService;

	@PostConstruct
	public void init() {
		User user1 = new User("99999990A", "Pedro", "Díaz");
		user1.setPassword("123456");
		user1.setRole(rolesService.getRoles()[0]);
		User user2 = new User("99999991B", "Lucas", "Núñez");
		user2.setPassword("123456");
		user2.setRole(rolesService.getRoles()[0]);
		User user3 = new User("99999992C", "María", "Rodríguez");
		user3.setPassword("123456");
		user3.setRole(rolesService.getRoles()[0]);
		User user4 = new User("99999993D", "Marta", "Almonte");
		user4.setPassword("123456");
		user4.setRole(rolesService.getRoles()[1]);
		User user5 = new User("99999977E", "Pelayo", "Valdes");
		user5.setPassword("123456");
		user5.setRole(rolesService.getRoles()[1]);
		User user6 = new User("99999988F", "Edward", "Núñez");
		user6.setPassword("123456");

		user6.setRole(rolesService.getRoles()[2]);
```

2.4 Actualización de controlador UserController

Formulario para User **GET /user/add,** obtiene la lista de roles y la envía a la vista para que pueda mostrarnos los roles disponibles.

Inyectamos el servicio **RolesService** en el controlador

```
@Controller
public class UsersController {

        @Autowired

        private RolesService rolesService;
```

En la función **GET /user/add** solicitamos la lista de roles y se la enviamos a la vista bajo el atributo **rolesList**. De esta forma la vista nos permitirá seleccionar un role.

```
@RequestMapping(value="/user/add")
public String getUser(Model model){
        model.addAttribute("rolesList", rolesService.getRoles());
        return "user/add";

}
```

El **POST /user/add** no es necesario modificarlo, ya que el objeto usuario se construirá automáticamente con los datos recibidos desde el formulario, basta con que reciba un dato con clave "role".

Sí que es necesario modificar el **POST /signup** poque ahora los usuarios deben tener un **role** y el formulario de signup no nos permite seleccionarlo, vamos a asignar a todos esos usuarios el ROLE_STUDENT.

```
@RequestMapping(value = "/signup", method = RequestMethod.POST)
public String signup(@ModelAttribute @Validated User user, BindingResult result, Model
model) {
    signUpFormValidator.validate(user, result);
    if (result.hasErrors()) {
        return "signup";
    }
    user.setRole(rolesService.getRoles()[0]);
    usersService.addUser(user);
    securityService.autoLogin(user.getDni(), user.getPasswordConfirm());
    return "redirect:home";
}
```

2.5 Actualización de vista /user/add

La vista **/user/add** debe mostrar los roles disponibles en la aplicación para que se puedan crear usuarios con diferentes roles.

Vamos a crear un nuevo campo en el formulario con nombre **role** (recordamos que para que el objeto se construya automáticamente al enviar el formulario el nombre del campo tiene que coincidir con el nombre del atributo en la clase User).

Creamos un componente <select> y recorremos la lista con los roles, **rolesList** , para cada role incluimos un elemento <option>. (**rolesList** es simplemente una lista de cadenas de texto)

```
<div class="container">
  <h2>Agregar usuario</h2>
  <form class="form-horizontal" method="post" action="/user/add">
        <div class="form-group">
                <label class="control-label col-sm-2" for=”role”>Role:</label>
                <div class="col-sm-10">
                        <select id="role" class="form-control" name="role">
                                <option th:each="role : ${rolesList}"
                                        th:value="${role}"
                                        th:text="${role}">
                                </option>
                 </select>
                </div>
        </div>
```

Importante: también nos faltaría especificar un **password** para los usuarios que se añaden desde **/user/add**, añadimos un nuevo campo con nombre **password**

```
    <div class="form-group">
        <label class="control-label col-sm-2" for="password">Password:</label>
        <div class="col-sm-10">
            <input type="password" class="form-control" name="password"
                placeholder="Introduzca el Password" />
        </div>
    </div>
```

2.6 Carga del Role en la autenticación

Modificamos el servicio **UserDetailsServiceImp,** en lugar de utilizar un role fijo cargamos el role asociado al usuario.

```
@Override
public UserDetails loadUserByUsername(String dni) throws UsernameNotFoundException{
    User user = usersRepository.findByDni(dni);

    Set<GrantedAuthority> grantedAuthorities = new HashSet<>();

    grantedAuthorities.add(new SimpleGrantedAuthority("ROLE_ESTUDIANTE"));

     grantedAuthorities.add(new SimpleGrantedAuthority(user.getRole()));

    return new org.springframework.security.core.userdetails.User(
            user.getDni(), user.getPassword(), grantedAuthorities);
}
```

3 Control de acceso por roles

Para definir el control de acceso y la autorización en spring web security debemos configurar el servicio de seguridad utilizando un conjunto de métodos. Con estos métodos permitimos o denegamos el acceso a usuarios, estas acciones pueden especificarse en función de roles específicos (si es que la aplicación tiene roles, ya que algunas aplicaciones muy simples no los tienen).

Algunos metodos comunmente usados son: **hasRole, hasAnyRole hasAuthority, hasAnyAuthority, permitAll, denyAll isAnonymous, isRememberMe, isAuthenticated**[1]**, isFullyAuthenticated**[2] **, authentication**, etc.

3.1.1 Configuración: autorización y control de acceso

Para definir el control de acceso tenemos que modificar el método ***configure*** la configuración de ***WebSegurityConfig*** e indicar las URLs a las que tienen acceso los usuarios autenticados.

Vamos a implementar la configuración relativa a las **notas**

- Los usuarios con cualquier autoridad pueden consultar la lista de notas, esto implica las URLS **/mark/list , /mark/update y /mark/details/***
- Solo un ROLE_PROFESSOR puede añadir **/mark/add** , eliminar **/mark/delete/*** y modificar **/mark/edit/*** notas.

Para indicar varios roles, debemos usar el metodo ***hasAnyAuthority()*** para indicar un único rol, podemos usar el método ***hasAuthority().***

URLs: las URLS de **detalles, eliminar y modificar**, incluyen parámetros, son de tipo **/mark/edit/34** , para especificar URLS utilizamos expresiones regulares con ** **/mark/edit/**** esto nos servirá para especificar cualquier URL que comience por **/mark/edit/**

De especifico a genérico: cuando especificamos los permisos debemos empezar por los permisos más **específicos** (por ejemplo, los relativos a ROLE_PROFESSOR, puede añdir, modificar y borrar)

Acabamos por los permisos más **genéricos,** como el resto de URLs relativas a mark son varias (/mark/list, /mark/update, /mark/details/*) las podemos encapsular en la expresión regular **/mark/****

```
@Override
protected void configure(HttpSecurity http) throws Exception {
    http
        .csrf().disable()
        .authorizeRequests()
            .antMatchers("/css/**", "/img/**", "/script/**", "/", "/signup",”/login/**”).permitAll()
            .antMatchers("/mark/add").hasAuthority("ROLE_PROFESSOR")
```

[1] Devuelve True si el usuario principal no es anónimo (Está autenticado)

[2] Devuelve True si el usuario principal se ha autenticado o es un usuario remember-me (credenciales guardada en el navegador)

```
            .antMatchers("/mark/edit/*").hasAuthority("ROLE_PROFESSOR")
            .antMatchers("/mark/delete/*").hasAuthority("-ROLE_PROFESSOR")
            .antMatchers("/mark/**").hasAnyAuthority("ROLE_STUDENT","ROLE_PROFESSOR","ROLE_ADMIN")
            .anyRequest().authenticated()
               .and()
        .formLogin()
            .loginPage("/login")
            .permitAll()
```

El orden de declaración es muy importante. Sí hubiésemos añadido en primer lugar **.antMatchers("/mark/**").hasAnyAuthority("ROLE_STUDENT","ROLE_PROFESSOR","ROLE_ADMIN")**

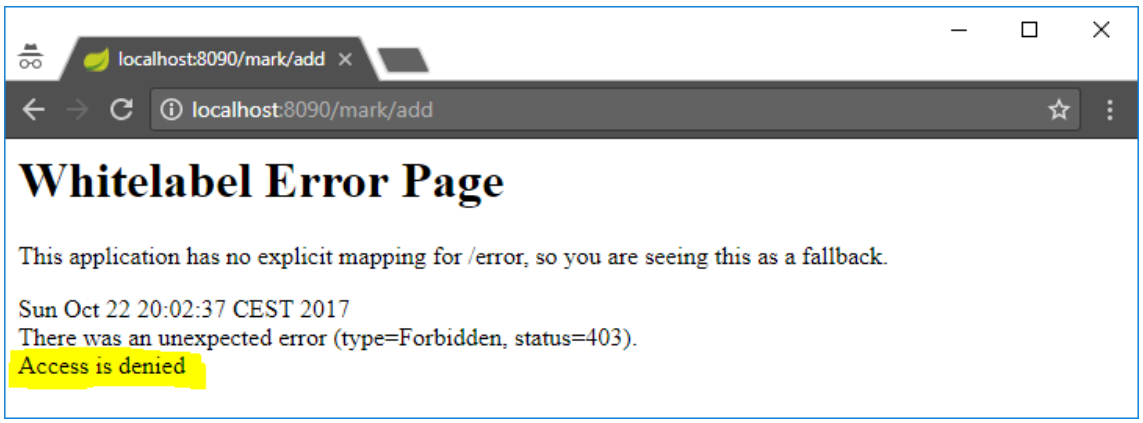
Estos perfiles tendrían acceso a todas las URLS que empiezan con **/mark/** incluyendo las **/mark/edit/*** , etc.

Implementamos ahora la confugiración relativa a usuarios

- Solo un ROLE_ADMIN puede acceder a todo lo relativo a la gestión de usuarios **/user/****

```
.authorizeRequests()
     .antMatchers("/css/**", "/img/**", "/script/**", "/", "/signup","/login/**").permitAll()
     .antMatchers("/mark/add").hasAuthority("ROLE_PROFESSOR")
     .antMatchers("/mark/edit/*").hasAuthority("ROLE_PROFESSOR")
     .antMatchers("/mark/delete/*").hasAuthority("ROLE_PROFESSOR")
     .antMatchers("/mark/**").hasAnyAuthority("ROLE_STUDENT","ROLE_PROFESSOR","ROLE_ADMIN")
     .antMatchers("/user/**").hasAnyAuthority("ROLE_ADMIN")
     .anyRequest().authenticated()
```

Ejecutamos la aplicación y nos autenticamos con DNI: 99999990A passwword: 123456 (tiene ROLE_STUDENT, vamos a probar modificar una nota, la aplicación nos muestra un error de acceso ***"Access is Denied".***



Nota: ***Roles y authorities*** son similares en Spring. La diferencia es que, apartir de spring security 4, **Roles** tiene un significado más semántico. Los roles tienen que comenzar con el prefijo ROLE_, si usamos el método hasRole() este lo añade automáticamente. Así que, ***hasAuthority ("ROLE_ PROFESSOR")*** es similar a ***hasRole("PROFESSOR")***.

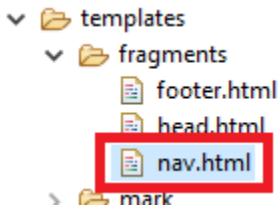
3.1.2 Vistas dependientes de roles

Vamos a obtener el **role** del usuario y mostrar unos elementos u otros en la vista en función del mismo, por ejemplo, es muy común que las opciones del menú de navegación sean diferentes para cada role.

Como comentamos anteriormente ***thymeleaf*** tiene muy buena integración con spring security (gracias a la dependencia ***thymeleaf-extras-springsecurity4*** que añadimos anteriormente).

Comenzamos modificando el **/fragments/nav.html** en funcion de la autorización del usuario se muestre un contenido u otro.

- ***sec:authorize="isAuthenticated()"*** nos permite saber si el usuario está autenticado.
- ***sec:authorize="hasRole('ROLE_ADMIN')"*** nos permite saber si el usuario autorizado tiene un rol específico.



- El contenido de la nav se muestran solamente si el usuario está autenticado ***sec:authorize="isAuthenticated()"***
- En el menú desplegable de "gestión de notas" la opción de "Agregar Nota" solo se muestra si el usuario autenticado tiene ROLE_PROFESSOR ***sec:authorize="hasRole('ROLE_PROFESSOR')"***
- El menú desplegable de "gestión de usuarios" se muestra solo si el usuario autenticado tiene ROLE_ADMIN, ***sec:authorize="hasRole('ROLE_ADMIN')"***

```
<!-- nav.html -->
<nav class="navbar navbar-inverse">
     <div class="container-fluid">
          <div class="navbar-header">
               <button type="button" class="navbar-toggle"
                    data-toggle="collapse" data-target="#myNavbar">
                    <span class="icon-bar"></span>
                    <span class="icon-bar"></span>
                    <span class="icon-bar"></span>
               </button>
               <img src="/img/student-48.png" alt="logo" />
          </div>
          <div class="collapse navbar-collapse" id="myNavbar"  >
               <ul class="nav navbar-nav" sec:authorize="isAuthenticated()">
                    <li><a href="/">Home</a></li>
                    <li id= "marks-menu" class="dropdown">
                       <a class="dropdown-toggle" data-toggle="dropdown" href="#">
                           Gestión de notas  <span class="caret"></span>
                       </a>
```

```
                    <ul class="dropdown-menu">
                        <li sec:authorize="hasRole('ROLE_PROFESSOR')" >
                            <a href="/mark/add">Agregar Nota</a>
                        </li>
                        <li><a href="/mark/list">Ver Notas</a></li>
                    </ul>
                </li>
                <li id= "users-menu" class="dropdown"
                    sec:authorize="hasRole('ROLE_ADMIN')" >
                    <a class="dropdown-toggle" data-toggle="dropdown" href="#">
                        Gestión de Usuarios <span class="caret"></span>
                    </a>
                    <ul class="dropdown-menu">
                      <li><a href="/user/add">Agregar usuario</a></li>
                      <li><a href="/user/list">Ver Usuarios</a></li>
                    </ul>
                </li>
```

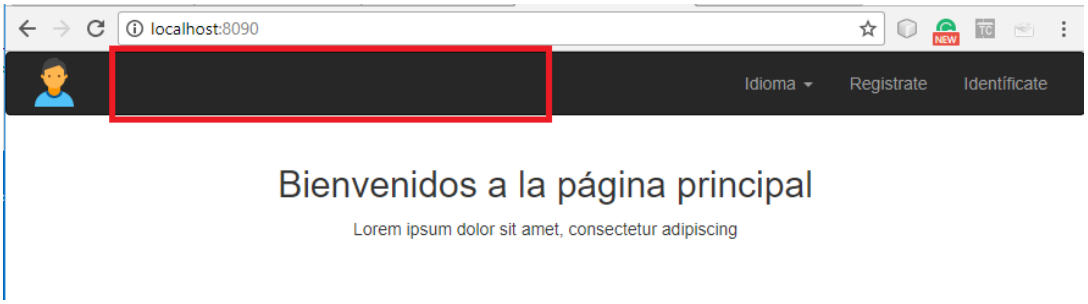
- Mostramos la opción de **Identificar** y **Registrarse** solo si no hay un usuario autenticado, si hay un usuario autenticado mostramos la opción de **Desconectar**.

```
                <li sec:authorize="!isAuthenticated()">
                     <a href="/signup" th:text="#{signup.message}">
                         <span class="glyphicon glyphicon-user"></span>
                         Registrate
                     </a>
                 </li>
                 <li sec:authorize="!isAuthenticated()">
                     <a href="/login" th:text="#{login.message}">
                         <span class="glyphicon glyphicon-log-in"></span>
                         Identifícate
                     </a>
                 </li>
                 <li sec:authorize="isAuthenticated()">
                     <a href="/logout" >
                         <span class="glyphicon glyphicon-log-out"></span>
                         Desconectar
                     </a>
                 </li>
            </ul>
        </div>
    </div>
</nav>
```

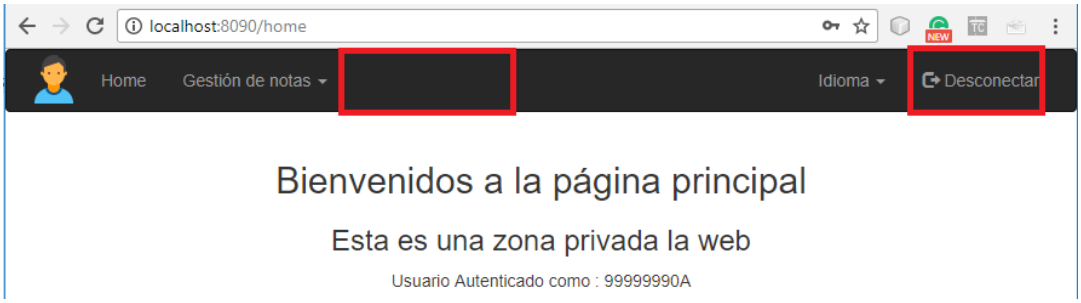
No hace falta implementar el controlador **/logout** ya que se trata de la URL estándar en Spring Security, cuando configuramos la seguridad en **WebSecurityConfig** especificamos que se permitia el logout.

```
        .formLogin()
             .loginPage("/login")
             .permitAll()
             .defaultSuccessUrl("/home")
             .and()
        .logout()
             .permitAll();
```

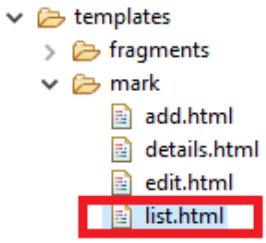
Probamos los cambios realizados en la aplicación



Nos identificamos con el usuario DNI: 99999990A y password: 123456 que tiene ROLE_STUDENT.



Modificamos la vista */mark/list* para que los enlaces de **editar** y **borrar** solo estén visibles para usuarios con **ROLE_PROFESSOR**.




```
<div class="container">
  <h2>Notas</h2>
  <p>Las notas que actualmente figuran en el sistema son las siguientes:</p>
    <button type="button" id="updateButton" class="btn btn-default">Actualizar</button>
  <script>
      $( "#updateButton" ).click(function() {
          $("#tableMarks").load('/mark/list/update');
      });
  </script>

  <div class="table-responsive">
  <table class="table table-hover" th:fragment="tableMarks" id="tableMarks">
    <thead>
      <tr>
        <th class="col-md-1">id</th>
        <th>Descripción</th>
        <th>Puntuación</th>
        <th class="col-md-1"> </th>
        <th class="col-md-1"> </th>
        <th class="col-md-1"> </th>
      </tr>
```

```
    </thead>
    <tbody>
      <tr th:each="mark : ${markList}">
        <td th:text="${mark.id}"> 1</td>
        <td th:text="${mark.description}"> Ejercicio 1</td>
        <td th:text="${mark.score}">10</td>
        <td><a th:href="${'/mark/details/' + mark.id}">detalles</a></td>
        <td><a sec:authorize="hasRole('ROLE_PROFESSOR')"
               th:href="${'/mark/edit/' + mark.id}">modificar</a></td>
        <td><a sec:authorize="hasRole('ROLE_PROFESSOR')"
               th:href="${'/mark/delete/' + mark.id}">eliminar</a></td>
      </tr>
    </tbody>
  </table>
  </div>
</div>
```

Ejecutamos la aplicación y comprobamos que a los usuarios con ROLE_STUDENT no se les les muestran los botones de **editar** y **eliminar**.

Nota: esta no es la unica vista de la aplicación en la que estamos mostrando los enlaces de editar y eliminar nota, deberiamos hacerlo condicionar al role en todas las partes (**/home**)

Finalmente podemos eliminar el uso de la etiqueta **[[${#httpServletRequest.remoteUser}]]** empleada en home.html, podemos utilizar la etiqueta ***sec:authentication="principal.username"*** (ambas obtienen la misma parámetro)

```
<div class="container" style="text-align: center">
        <h2 th:text="#{welcome.message}"></h2>
        <h3>Esta es una zona privada la web</h3>
        <p>
          Usuario Autenticado como :
          <span sec:authentication="principal.username"
                th:inline="text"> [[${#httpServletRequest.remoteUser}]] </span>
        </p>
```

Bienvenidos a la página principal

Esta es una zona privada la web

Usuario Autenticado como : 99999990A

Notas del usuario

3.1.3 Accion asociada a un alumno

Vamos a permitir que los alumnos puedan modificar una propiedad **reenvio** de sus notas, utilizarán esta propiedad para notificar que quieren reenviar el trabajo evaluado.

3.1.3.1 Actualizar entidad Mark

Incluimos la nueva propiedad en la entidad, por defecto tendrá el valor false.

```
@Entity
public class Mark {
        @Id
        @GeneratedValue
        private Long id;
        private String description;
        private Double score;
        private Boolean resend = false;
```

Implementamos los métodos get y set.

```
public Boolean getResend() {
        return resend;
}
public void setResend(Boolean resend) {
        this.resend = resend;

}
```

3.1.3.2 Actualizar MarkRepository

Implementamos un nuevo metodo que actualice la propiedad **revised** de la nota. Los métodos que modifican registros deben incluir la anotación @Modifying además la operación debería realizarse de forma transaccional @Transactional

```
public interface MarksRepository extends CrudRepository<Mark, Long>{
        @Modifying
        @Transactional
        @Query("UPDATE Mark SET resend = ?1 WHERE id = ?2")
        void updateResend(Boolean resend, Long id);

}
```

3.1.3.3 Actualizar MarkService

En el servicio incluimos una nueva función que nos permita modificar la propiedad **revised**, este servicio hara uso del repositorio implementado anteriormente.

```
public void setMarkResend(boolean revised,Long id){
        marksRepository.updateResend(revised, id);
}
```

3.1.3.4 Controlador MarksController

Incluimos respuesta a la URL **/mark/{id}/resend** para poner a **true** el atributo resend de una nota y otra URL /**mark/{id}/noresend**. para ponerlo a **false**

```
@RequestMapping(value="/mark/{id}/resend", method=RequestMethod.GET)
public String setResendTrue(Model model, @PathVariable Long id){
        marksService.setMarkResend(true, id);
        return "redirect:/mark/list";
}

@RequestMapping(value="/mark/{id}/noresend", method=RequestMethod.GET)
public String setResendFalse(Model model, @PathVariable Long id){
        marksService.setMarkResend(false, id);
        return "redirect:/mark/list";
}
```

3.1.3.5 Actualizar vista /mark/list

Sí el usuario esta autenticado como ROLE_STUDENT mostramos el atributo **resend** y la opción los dos enlaces para poder modificarlo.

```
<!DOCTYPE html>
<html lang="en">

<head th:replace="fragments/head"/>

<body>

<!-- Barra de Navegación superior -->
<nav th:replace="fragments/nav"/>

<div class="container">
  <h2>Notas</h2>
  <p>Las notas que actualmente figuran en el sistema son las siguientes:</p>
    <button type="button" id="updateButton" class="btn btn-default">Actualizar</button>
  <script>
       $( "#updateButton" ).click(function() {
           $("#tableMarks").load('/mark/list/update');
       });
  </script>

  <div class="table-responsive">
  <table class="table table-hover" th:fragment="tableMarks" id="tableMarks">
    <thead>
      <tr>
        <th class="col-md-1">id</th>
        <th>Descripción</th>
        <th>Puntuación</th>
        <th class="col-md-1"> </th>
        <th class="col-md-1"> </th>
```

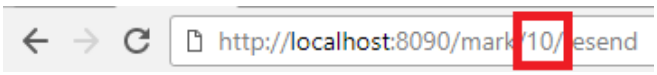
```
        <th class="col-md-1"> </th>
      </tr>
    </thead>
    <tbody>
      <tr th:each="mark : ${markList}">
        <td th:text="${mark.id}"> 1</td>
        <td th:text="${mark.description}"> Ejercicio 1</td>
        <td th:text="${mark.score}">10</td>
        <td><a th:href="${'/mark/details/' + mark.id}">detalles</a></td>
        <td><a sec:authorize="hasRole('ROLE_PROFESSOR')"
               th:href="${'/mark/edit/' + mark.id}">modificar</a>

            <div sec:authorize="hasRole('ROLE_STUDENT')" >
                <div th:if="${mark.resend}">
                    <a th:href="${'/mark/' + mark.id + '/noresend'}">Reenviar</a>
                </div>
                <div th:unless="${mark.resend}">
                    <a th:href="${'/mark/' + mark.id + '/resend'}">No reenviar</a>
                </div>
            </div>
        </td>
        <td><a sec:authorize="hasRole('ROLE_PROFESSOR')"
               th:href="${'/mark/delete/' + mark.id}">eliminar</a>
        </td>
      </tr>
```

Ejecutamos la aplicación, nos identificamos con DNI: 99999990A y password: 123456 accedemos a la lista de notas y comprobamos el funcionamiento.

id	Descripción	Puntuación		
1	Nota A1	10.0	detalles	No reenviar
2	Nota A2	9.0	detalles	Reenviar
3	Nota A3	7.0	detalles	No reenviar
4	Nota A4	6.5	detalles	Reenviar
5	Nota B2	4.3	detalles	No reenviar
6	Nota B1	5.0	detalles	Reenviar
7	Nota B3	8.0	detalles	No reenviar
8	Nota B4	3.5	detalles	No reenviar

3.1.3.6 Comprobacion de seguridad: propietario del recurso

Es importante validar que el usuario que esta modificando la propiedad **reenvio** es el propietario de la nota, de lo contrario cualquier usuario que conozca la id de la nota podría cambiar el valor de la propiedad.



Debemos hacer comprobaciones adicionales en el servicio **MarksService**, concretramente en la función **setMarkResend()**.

Utilizamos el **SecurityContextHolder** para obtener los datos del usuario autenticado, el name del usuario autenticado se corresponde con el **dni**. Obtenemos la nota que se esta

intentando modificar y comprobamos si el **dni** de su usuario coincide con el **dni** del usuario autenticado, solo en ese caso realizamos la modificación.

```
public void setMarkResend(boolean revised,Long id){
        Authentication auth = SecurityContextHolder.getContext().getAuthentication();
        String dni = auth.getName();

        Mark mark = marksRepository.findOne(id);

        if( mark.getUser().getDni().equals(dni)  ) {
                marksRepository.updateResend(revised, id);
        }
}
```

Ejecutamos la aplicación y comprobamos que no podemos modificar notas que no pertecen al usuario autenticado.

3.1.4 (Extra-avanzado) Actualización de la tabla como fragmento

La tabla en la que se muestran las notas en la vista **/mark/list** estaba identificada como un fragmento.

```
<div class="table-responsive">
  <table class="table table-hover" th:fragment="tableMarks" id="tableMarks">
    <thead>
      <tr>
        <th class="col-md-1">id</th>
        <th>Descripción</th>
        <th>Puntuación</th>

                <th class="col-md-1"> </th>
```

En practicas anteriores habíamos visto un ejemplo sobre como actualizar únicamente un fragmento de una vista, concretamente esta funcionalidad se encontraba en el botón update de esta misma vista. La URL **/mark/list/update** retorna únicamente el fragmento de la tabla actualizado, con la función load de javascript insertábamos el retorno de esa URL en un el componente HTML de la página con id **tableMarks**

```
  <p>Las notas que actualmente figuran en el sistema son las siguientes:</p>
  <button type="button" id="updateButton" class="btn btn-default">Actualizar</button>
  <script>
          $( "#updateButton" ).click(function() {
                  $("#tableMarks").load('/mark/list/update');
          });
  </script>

  <div class="table-responsive">
  <table class="table table-hover" th:fragment="tableMarks" id="tableMarks">
    <thead>

      <tr>
```

Vamos a hacer que los botones de **reenvio** y **no reenvio** actualicen solamente el fragmento de la tabla, en lugar de toda la vista.

Puntos a tener en cuenta.

- **Botones**, debemos crear los botones de **reenvio** y **no reenvio** para cada nota, por lo tanto, cada botón debe tener una id diferente, podemos utilizar el **mark.id** para

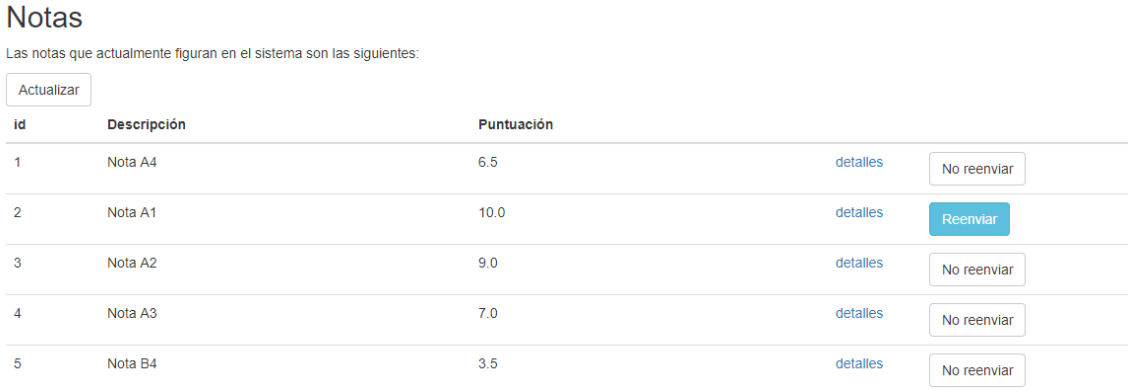
crear los id de los botones. La propiedad **th:id** nos permite asignar ids combinando texto y propiedades, por ejemplo: **th:id="${'resendButton' + mark.id}"**

- **Fragmentos de Script declaración:** necesitamos crear unos script que relicen las peticiones a las URLs: **mark/{id}/resend** y **/mark/{id}/noresend,** los scripts que utilizan elementos de Thymeleaft deben ser declarados de una forma muy especifica, incluyendo la etiqueta <script **th:inline="javascript"**> y encapsulando todo el script en un **/*<![CDATA[*/**
- **Fragmentos de Srcipt, atributos thymeleaft:** para insertar el valor de un atributo thymeleaft en un script utilizamos la sintaxis **[[${<nombre_atributo>}]]**, por ejemplo: **[[${mark.id}]]**
- **Fragmentos de Script, peticiones:** para realizar una petición get en jQuery usamos la función $.get(URL, función de callback). La función de callback se ejecuta cuando la petición finaliza, una vez finalizada la petición get que realizaremos para cambiar el valor del atributo resend, podemos volver a cargar el fragmento de la tabla.

```
<tbody>
  <tr th:each="mark : ${markList}">
    <td th:text="${mark.id}"> 1</td>
    <td th:text="${mark.description}"> Ejercicio 1</td>
    <td th:text="${mark.score}">10</td>
    <td><a th:href="${'/mark/details/' + mark.id}">detalles</a></td>
    <td><a sec:authorize="hasRole('ROLE_PROFESSOR')"
           th:href="${'/mark/edit/' + mark.id}">modificar</a>

        <div sec:authorize="hasRole('ROLE_STUDENT')" >
            <div th:if="${mark.resend}">
                <a th:href="${'/mark/' + mark.id + '/noresend'}">Reenviar</a>

                <button type="button" th:id="${'resendButton' + mark.id}"
                        class="btn btn-info">Reenviar</button>
                <script th:inline="javascript">
                  /*<![CDATA[*/
                  $( "#resendButton[[${mark.id}]]" ).click(function() {
                      $.get( "/mark/[[${mark.id}]]/noresend", function( data ) {
                            $("#tableMarks").load('/mark/list/update');
                      });
                  });
                  /*]]>*/
                </script>
            </div>
            <div th:unless="${mark.resend}">
                <a th:href="${'/mark/' + mark.id + '/resend'}">No reenviar</a>

                <button type="button" th:id="${'noresendButton' + mark.id}"
                        class="btn btn-default">No reenviar</button>
                <script th:inline="javascript">
                  /*<![CDATA[*/
                  $( "#noresendButton[[${mark.id}]]" ).click(function() {
                      $.get( "/mark/[[${mark.id}]]/resend", function( data ) {
                            $("#tableMarks").load('/mark/list/update');
                      });
                  });
                  /*]]>*/
                </script>
            </div>
        </div>
```

```
    </td>
    <td><a sec:authorize="hasRole('ROLE_PROFESSOR')"
```

Ejecutamos la aplicación y comprobamos el resultado.



4 Consultas

En esta sección veremos como podemos realizar consultas a la base de datos en función de unos criterios y mostrar únicamente los resultados que coincidan con ese criterio

Los repositorios JPA admiten la definición de consultas de forma sencilla. Una opción (aunque no la única) para realizar estas consultas es utilizar la anotación ***@Query***.

4.1 Consultar notas

Actualmente la aplicación muestra todas las notas en **/mark/list** . Vamos a modificar la aplicación para:

- **Mostrar las notas solo del usuario autenticado**, si este tiene ROLE_STUDENT
- **Mostrar todas las notas de la aplicación** si el usuario autenticado tiene ROLE_PROFESSOR

4.1.1 Actualizar MarksRepository

Accedemos a ***MarksRepository*** y añadimos el metodo **findAllByUser(User user)**. Al tratarse de una función se sue el patrón de nombrado **findAllBy<parámetro>** ni siquiera es necesario especificar la consulta SQL con el atributo @Query.

```
public interface MarksRepository extends CrudRepository<Mark, Long>{

        List<Mark> findAllByUser(User user);
```

Sí que seria necesario utilizer el atributo **@Query** si siquieramos hacer una consulta más especifica que no sea simplemente un select de un parámetro, por **ejemplo** solo las notas con un valor de 5 o más.

```
        @Query("SELECT r FROM Mark r WHERE r.user = ?1 AND r.score >= 5 ")
        List<Mark> findAllPassedByUser(User user);
```

4.1.2 Actualizar MarksService

Accedemos a ***MarksService***, incluimos un servicio que reciba un **Usuario** como parámetro, en función del ROLE del usuario retornará todas las notas de la aplicación (ROLE_PROFESSOR) o solo las notas relativas al usuario (ROLE_STUDENT).

```
public List<Mark> getMarksForUser (User user){
        List<Mark> marks = new ArrayList<Mark>();
        if ( user.getRole().equals("ROLE_STUDENT")) {
                marks = marksRepository.findAllByUser(user);
        }
        if ( user.getRole().equals("ROLE_PROFESSOR")){
                marks = getMarks();
        }
        return marks;
}
```

4.1.3 Modificar MarksController

Accedemos a ***MarkController*** y modificamos la respuesta a la URL **/mark/list.** Obtenemos el usuario autenticado podemos hacerlo incluyendo la variable **Principal** como parámetro del método. La varaible principal tiene el nombre de la autenticación (que coincide con el dni del usuario), obtenemos el usuario al que pertenece el DNI y llamamos al servicio ***getMarksForUser (user);***

```
@RequestMapping("/mark/list")
public String getList(Model model, Principal principal){
        String dni = principal.getName(); // DNI es el name de la autenticación
        User user = usersService.getUserByDni(dni);
        model.addAttribute("markList", marksService.getMarksForUser(user) );
        return "mark/list";
}
```

Modificamos de igual modo la respuesta a la URL **/mark/list/update** la cual se encarga de actualizar de forma dinámica la lista de notas.

```
@RequestMapping("/mark/list/update")
public String updateList(Model model, Principal principal){
        String dni = principal.getName(); // DNI es el name de la autenticación
        User user = usersService.getUserByDni(dni);
        model.addAttribute("markList", marksService.getMarksForUser(user)  );
        return "mark/list :: tableMarks";

}
```

Anteriormente habíamos visto otra forma de obtener el usuario autenticado, a traves del **SecurityContextHolder**, ambas son validas, no obstante el **SecurityContextHolder** se

puede utilizar desde cualquier punto de la aplicación, no solo desde los controladores.

```
@RequestMapping("/mark/list")
public String getList(Model model){
        Authentication auth = SecurityContextHolder.getContext().getAuthentication();
        String dni = auth.getName();
        User user = usersService.getUserByDni(dni);
        model.addAttribute("markList", marksService.getMarksForUser(user) );
        return "mark/list";

}
```

Guardamos los cambios y accedemos a la aplicación con los siguientes perfiles:

- DNI: 99999990A password: 123456 -> ROLE_STUDENT
- DNI: 99999977E password: 123456 -> ROLE_PROFESSOR

Sí accedemos a **/mark/list** se deberian mostrar solo las notas propias para el estudiante y todas las notas para el profesor.

La aplicación funciona correctamente pero observamos un comportamiento inadecuado cuando pulsamos el boton de **Reenviar**, se cambia el orden de los registros (JPA nos devuelve los registros por orden de modificación, cuando uno se modifica se retorna primero, esto altera el orden de las notas).

Podemos solucionar este problema especificando la **@Query** en el repositorio **MarksRepository**.

```
public interface MarksRepository extends CrudRepository<Mark, Long>{

        @Query("SELECT r FROM Mark r WHERE r.user = ?1 ORDER BY r.id ASC ")
        List<Mark> findAllByUser(User user);
```

5 Búsqueda

5.1 Búsqueda

Vamos a incluir un formulario de búsqueda parar poder encontrar **notas** en base a un criterio. Realizaremos una consulta en el repositorio y mostraremos las notas que cumplan el criterio, permitiendo realizar busquedas por: ***nombre, DNI*** o ***descripción.*** Las busquedas que vamos a realizar en esta funcionalidad van a ser parciales, el contenido buscado no va a tener porque coincidir exactamente con el del campo.

5.1.1 Actualizar MarksRepository

Modificamos el repositorio ***MarksRepository*** añadiendo los siguientes métodos:

- **searchByDescriptionAndName *(String searchtext)* ,** retorna notas de toda la aplicación cuando el texto buscado coincide con el nombre del usuario o la descripción de la nota.
- **searchByDescriptionNameAndUser *(String searchtext, User user)* ,** retorna notas relaccionadas con el usuario enviado como parámetro, cuando el texto buscado coincide con el nombre del usuario o la descripción de la nota.

Hemos pasado todos los textos a minúsculas con la función **LOWER** para que la aplicación no sea sensible a mayusculas/minúsculas.

```
public interface MarksRepository extends CrudRepository<Mark, Long>{

	@Query("SELECT r FROM Mark r WHERE (LOWER(r.description) LIKE LOWER(?1) OR
LOWER(r.user.name) LIKE LOWER(?1))")
	List<Mark> searchByDescriptionAndName(String seachtext);

	@Query("SELECT r FROM Mark r WHERE (LOWER(r.description) LIKE LOWER(?1) OR
LOWER(r.user.name) LIKE LOWER(?1))  AND r.user = ?2 ")
	List<Mark> searchByDescriptionNameAndUser(String seachtext, User user);
```

5.1.2 Actualizar MarksService

Actualizamos el servicio ***MarksService***, añadimos el metodo **searchMarksByDescriptionAndNameForUser (String searchText, User user)** se encargará de realizar una búsqueda en las notas, las notas del propio usuario si el usuario autenticado es ROLE_STUDENT, las notas de todos los usuarios si el usuario autenticado es ROLE_PROFESSOR.

```
public List<Mark> searchMarksByDescriptionAndNameForUser (String searchText, User user){
	List<Mark> marks = new ArrayList<Mark>();
	if ( user.getRole().equals("ROLE_STUDENT")) {
		marks = marksRepository.searchByDescriptionNameAndUser(searchText, user);
	}
	if ( user.getRole().equals("ROLE_PROFESSOR")){
		marks = marksRepository.searchByDescriptionAndName(searchText);
	}
	return marks;
}
```

5.1.3 Actualizar MarksController

Accedemos a **MarksController** y actualizamos el metodo que responde a la URL **/mark/list .** Vamos q permitir que esta URL reciba un parámetro con clave **searchText** que será opcional.

- Sí recibimos el parámetro **searchText** utilizamos el servicio **searchMarksByDescriptionAndNameForUser**
- Sí **no** recibimos ningun praémtro **searcText** utilizamos el servicio que utilizamos anteriormetne y que devuelve todas las notas **getMarksForUser**

```
@RequestMapping("/mark/list")
public String getList(Model model, Principal principal,
                @RequestParam(value = "", required=false) String searchText){

        String dni = principal.getName(); // DNI es el name de la autenticación
        User user = usersService.getUserByDni(dni);
        if (searchText != null && !searchText.isEmpty()) {
                model.addAttribute("markList",
                  marksService.searchMarksByDescriptionAndNameForUser (searchText, user) );
        } else {
                model.addAttribute("markList", marksService.getMarksForUser(user) );
        }

        return "mark/list";
}
```

5.1.4 Actualizar la vista mark/list

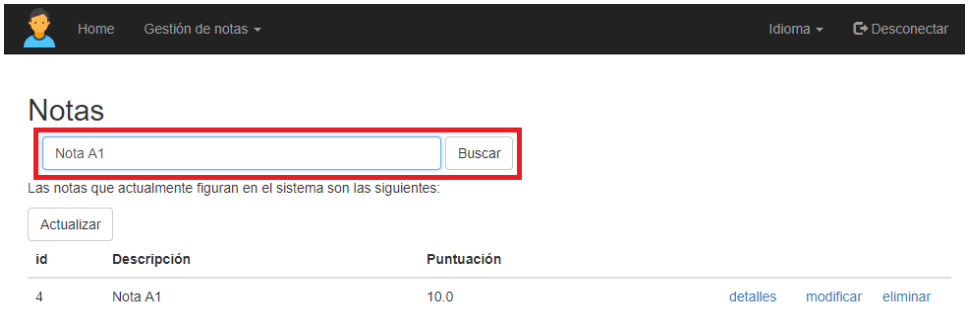
Ahora modificamos el fichero ***mark/list*** para incluir un formulario de búsqueda, se basará en unico campo con nombre **seachText** que enviará una petición GET contra **/mark/list**

```
<div class="container">
  <h2>Notas</h2>
  <form class="navbar-form" action="/mark/list">
        <div class="form-group">
            <input name="searchText" type="text" class="form-control" size="50"
                placeholder="Buscar por descripción o nombre del alumno">
        </div>
        <button type="submit" class="btn btn-default">Buscar</button>
 </form>
```

Guardamos los cambios y accedemos a la aplicación con los siguientes perfiles:

- DNI: 99999990A password: 123456 -> ROLE_STUDENT
- DNI: 99999977E password: 123456 -> ROLE_PROFESSOR

Las busquedas deben realizarse con cadenas exactas.



5.1.5 Buscar por cadena contenida

Sí quisiésemos que la búsqueda comprobase únicamente si la cadena buscada se encuentra en la descripción o el nombre del usuario (sin necesidad de que sea una coincidencia exacta) podríamos utilizar comodines de SQL **%searchText%. Accedemos al MarkService** e incluimos los comodines en el método **searchMarksByDescriptionAndNameForUser**

```
public List<Mark> searchMarksByDescriptionAndNameForUser (String searchText, User user){
        List<Mark> marks = new ArrayList<Mark>();
        searchText = "%"+searchText+"%";
        if ( user.getRole().equals("ROLE_STUDENT")) {
                marks = marksRepository.searchByDescriptionNameAndUser(searchText, user);
        }
        if ( user.getRole().equals("ROLE_PROFESSOR")){
                marks = marksRepository.searchByDescriptionAndName(searchText);
        }
        return marks;

}
```

5.1.6 Ejercicio propuesto

Implementar sistema de búsquedas en la gestión de usuarios.

6 Paginación

Cuando una lista contiene muchos elementos no suele ser aconsejable mostrarlos en una única página. Tampoco es eficiente solicitar al servidor/base de datos listas con miles o millones de recursos que realmente no van a ser utilizados. Por eso, es importante tener un sistema de paginación, vamos a incluir un sistema de paginación para visualizar las notas.

6.1 Actualizar MarksRepository

Actualmente, los métodos definidos en ***MarksRepository*** devuelven listas de objetos de notas (marks), como por ejemplo **List<Mark> findAllByUser(User user);**

A partir de ahora, para devolver una lista de notas paginadas utilizaremos el paquete ***org.springframework.data.domain.Page,*** solo tenemos que modificar el retorno de los métodos y utilizar ***Page<Mark>.***

Todos los métodos van a recibir además un parámetro adicional **Pageable**

Tambien tenemos que incluir el método **findAll()** (a pesar de que el CrudRepository ya nos lo da implementado) porque por defecto este método no retorna un **Page<Mark>**

Asegurarse de importar la clase **Pageable** correcta: import **org.springframework.data.domain.Pageable**;

```
import org.springframework.data.domain.Page;
import org.springframework.data.domain.Pageable;
import org.springframework.data.jpa.repository.Modifying;
import org.springframework.data.jpa.repository.Query;
import org.springframework.data.repository.CrudRepository;
import org.springframework.transaction.annotation.Transactional;

import com.uniovi.entities.Mark;
import com.uniovi.entities.User;

public interface MarksRepository extends CrudRepository<Mark, Long>{

	@Query("SELECT r FROM Mark r WHERE (LOWER(r.description) LIKE LOWER(?1) OR
LOWER(r.user.name) LIKE LOWER(?1))")
	Page<Mark> searchByDescriptionAndName(Pageable pageable, String seachtext);

	@Query("SELECT r FROM Mark r WHERE (LOWER(r.description) LIKE LOWER(?1) OR
LOWER(r.user.name) LIKE LOWER(?1))  AND r.user = ?2 ")
	Page<Mark> searchByDescriptionNameAndUser(Pageable pageable, String seachtext, User user);

	@Query("SELECT r FROM Mark r WHERE r.user = ?1 ORDER BY r.id ASC ")
	Page<Mark> findAllByUser(Pageable pageable, User user);

	Page<Mark> findAll(Pageable pageable);

	@Modifying
	@Transactional
	@Query("UPDATE Mark SET resend = ?1 WHERE id = ?2")
	void updateResend(Boolean resend, Long id);
```

6.2 Actualizar MarksService

Ahora tenemos que modificar en el servicio ***MarksService*** encargado de gestionar las notas.

Modificamos los metodos que retornaban lista de notas, en lugar de retornar ***List<Mark>*** debe utilizar ***Page<Mark>***, también deben recibir un parámetro **pageable.**

Asegurarse de importar la clase correcta: import **org.springframework.data.domain.Pageable**;

```
@Service
public class MarksService {

	@Autowired
	private HttpSession httpSession;

	@Autowired
	private MarksRepository marksRepository;


	public Page<Mark> getMarksForUser(Pageable pageable, User user){
		Page<Mark> marks = new PageImpl<Mark>(new LinkedList<Mark>());
		if ( user.getRole().equals("ROLE_STUDENT")) {
			marks = marksRepository.findAllByUser(pageable, user);
		}
		if ( user.getRole().equals("ROLE_PROFESSOR")){
			marks = getMarks(pageable);
		}
		return marks;
	}


	public Page<Mark> searchMarksByDescriptionAndNameForUser (Pageable pageable,
			String searchText, User user){

		Page<Mark> marks = new PageImpl<Mark>(new LinkedList<Mark>());
		searchText = "%"+searchText+"%";
		if ( user.getRole().equals("ROLE_STUDENT")) {
			marks = marksRepository.
				searchByDescriptionNameAndUser(pageable, searchText, user);
		}
		if ( user.getRole().equals("ROLE_PROFESSOR")){
			marks = marksRepository.
				searchByDescriptionAndName(pageable, searchText);
		}
		return marks;
	}



	public Page<Mark> getMarks(Pageable pageable){
		Page<Mark> marks = marksRepository.findAll(pageable);
		return marks;
	}
```

6.3 Modificar MarksController

Debemos modificar en el controlador **MarksController**, para que todos los métodos que devuelven una ***List<Mark>*** devuelvan una lista paginada ***Page<Mark>.*** El objeto **Page<Mark>** contiene mucha información sobre el sistema de paginación (página actual, totales, etc.) y también la lista con las notas, podemos acceder a ella utilizando el método **getContent(); Utilizando esta función obtenemos la lista de notas que las vistas están esperando.**

Los métodos afectados por la paginación también deberán recibir un parámetro ***Pageable pageable***, este parámetro sirve para gestionar en las URL los parámetros **page** y **size** , http://localhost:8090/mark/list?page=1&size=5 el parámetro size es opcional si no lo incluimos usa uno por defecto.

Asegurarse de importar la clase correcta: import **org.springframework.data.domain.Pageable**;

```
@RequestMapping("/mark/list")
public String getList(Model model, Pageable pageable, Principal principal,
                @RequestParam(value = "", required=false) String searchText){

        String dni = principal.getName(); // DNI es el name de la autenticación
        User user = usersService.getUserByDni(dni);
        Page<Mark> marks = new PageImpl<Mark>(new LinkedList<Mark>());
        if (searchText != null && !searchText.isEmpty()) {
                marks =  marksService
                        .searchMarksByDescriptionAndNameForUser(pageable, searchText, user);

        } else {
                marks = marksService.getMarksForUser(pageable, user) ;
        }

        model.addAttribute("markList", mark.getContent());

        return "mark/list";

}

        @RequestMapping("/mark/list/update")
        public String updateList(Model model, Pageable pageable, Principal principal){
                String dni = principal.getName(); // DNI es el name de la autenticación
                User user = usersService.getUserByDni(dni);
                Page<Mark> marks = marksService.getMarksForUser(pageable, user);
                model.addAttribute("markList", marks.getContent() );
                return "mark/list :: tableMarks";

        }
```

6.4 Actualizar la Configuración de la aplicación

Para que Spring sepa cómo convertir los parámetros GET **page** y **size** a objetos **Pageable**, debemos configurar un ***HandlerMethodArgumentResolver***. Spring Data proporciona un ***PageResearchResolver***, pero utiliza la antigua interfaz ***ArgumentResolver*** en lugar de la nueva interfaz (Spring 3.1)

HandlerMethodArgumentResolver.[3] Debemos añadir un nuevo método ***addArgumentResolvers*** a la clase **CustomConfiguration**.

Indicamos también los valores por defecto para Page y size que serán 0 y 5. (page=0 es e la primera).

```
package com.uniovi;
import java.util.List;
import java.util.Locale;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import org.springframework.web.servlet.LocaleResolver;
import org.springframework.web.servlet.config.annotation.InterceptorRegistry;
import org.springframework.web.servlet.config.annotation.WebMvcConfigurerAdapter;
import org.springframework.web.servlet.i18n.LocaleChangeInterceptor;
import org.springframework.web.servlet.i18n.SessionLocaleResolver;
import org.springframework.data.domain.PageRequest;
import org.springframework.data.web.PageableHandlerMethodArgumentResolver;
import org.springframework.web.method.support.HandlerMethodArgumentResolver;

@Configuration
public class CustomConfiguration extends  WebMvcConfigurerAdapter{

	@Override
	public void addArgumentResolvers(List<HandlerMethodArgumentResolver> argumentResolvers) {

		PageableHandlerMethodArgumentResolver resolver =
				new PageableHandlerMethodArgumentResolver();

		resolver.setFallbackPageable(new PageRequest(0, 5));
		argumentResolvers.add(resolver);
		super.addArgumentResolvers(argumentResolvers);
	}
```

En este el sistema de paginación ya estaría activo (aunque todavía nos falta incluir los botones de acceso a las páginas en la vista), podemos probarlo entrando con el perfil profesor (ya que puede visualizar muchas notas):

DNI: 99999977E password: 123456 -> ROLE_PROFESSOR

Podemos probar varias URLs:

http://localhost:8090/mark/list?page=2&size=1

http://localhost:8090/mark/list?page=2 (si no incluimos size usa uno por defecto 5).

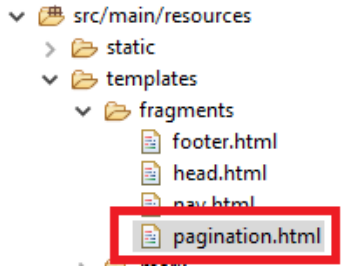
http://localhost:8090/mark/list?page=3&size=2

[3]https://www.javacodegeeks.com/2013/03/implement-bootstrap-pagination-with-spring-data-and-thymeleaf.html

Existe un problema con el botón de actualizar que resolveremos al editar las vistas, se debe a que las peticiones a **/mark/list/update** actualmente no reciben parámetros **page** y **size** (estábamos realizando estas peticiones desde JavaScript sin incluir ningun tipo de parámetro **page** ni size, habría que incluir los mismos que hay en la petición actual)

6.5 Actualización de las vistas

Vamos a crear un nuevo fragmento ***pagination.html*** con el selector de las páginas por si más adelante nos interesa incluir paginación en otras partes de la aplicación.



Esta vista va a recibir como parámetro el objeto **Page<Mark> bajo la clave page.**

Este objeto tiene la función **getNumber()** para obtener la página actual y **getTotalPages()** para obtener . Hay muchas formas de mostrar las opciones de paginaciónes, nosotros vamos a optar por mostrar la primera página, la ultima y las cercanas a la actual, seguiremos la siguiente estrategia:

- Mostramos la primera página page=0
- Mostramos la página anterior a la actual, solo si realmente la hay
- Mostramos la página actual
- Mostramos la página siguiente a la actual, solo si realmente la hay
- Mostramos la página final page=getTotalPages()

Las paginas empiezan a contar en 0 aunque eso puede no resultar muy lógico para el usuario por eso en las etiquetas de texto vamos a mostrar un número más del que realmente corresponde al enlace.

El enlace **1** lleva a http://localhost:8090/mark/list?page=0 , el enlace **2** lleva a la http://localhost:8090/mark/list?page=1 , asi sucesivamente…

```
<div class="text-center">
    <ul class="pagination pagination-centered">
        <!-- Primera -->
        <li class="page-item" >
            <a class="page-link" th:href="@{'?page=0'}">Primera</a>
        </li>

        <!-- Anterior (si la hay ) -->
        <li class="page-item" th:if='${page.getNumber()-1 >= 0}'>
            <a class="page-link" th:href="@{'?page='+${page.getNumber()-1} }"
                th:text="${page.getNumber()}"></a>
        </li>

        <!-- Actual  -->
        <li class="page-item active" >
            <a class="page-link" th:href="@{'?page='+${page.getNumber()} }"
                th:text="${page.getNumber()+1}"></a>
        </li>

        <!-- Siguiente (si la hay)  -->
        <li class="page-item" th:if='${page.getNumber()+1 <= page.getTotalPages()-1}'>
            <a class="page-link" th:href="@{'?page='+${page.getNumber()+1} }"
                th:text="${page.getNumber()+2}"></a>
        </li>

        <!-- Última -->
        <li class="page-item" >
            <a class="page-link"
                th:href="@{'?page='+${page.getTotalPages()-1}}"> Última</a>
        </li>
    </ul>
</div>
```

Ahora incluimos el fragmento ***pagination.html*** en la vista ***/marks/list.html*** , lo hacemos en la parte final del contenedor principal

```
        </tbody>
  </table>
  </div>
  <footer th:replace="fragments/pagination"/>

</div>

<footer th:replace="fragments/footer"/>


</body>
```

Solamente nos falta que el **MarksController** en su respuesta a la petición **/mark/list** envie el parámetro **page** que se necesita para mostrar el sistema de paginación de forma correcta.

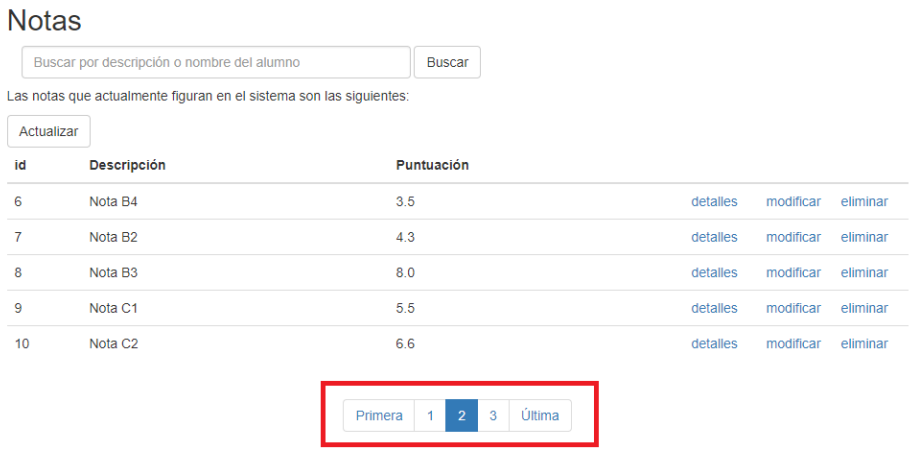
```
@RequestMapping("/mark/list")
public String getList(Model model, Pageable pageable, Principal principal,
                @RequestParam(value = "", required=false) String searchText){

        String dni = principal.getName(); // DNI es el name de la autenticación
        User user = usersService.getUserByDni(dni);
        Page<Mark> marks = new PageImpl<Mark>(new LinkedList<Mark>());
        if (searchText != null && !searchText.isEmpty()) {
                marks =  marksService
                        .searchMarksByDescriptionAndNameForUser(pageable, searchText, user);

        } else {
```

```
                marks = marksService.getMarksForUser(pageable, user) ;
        }

        model.addAttribute("markList", marks.getContent());
        model.addAttribute("page", marks);
        return "mark/list";

}
```

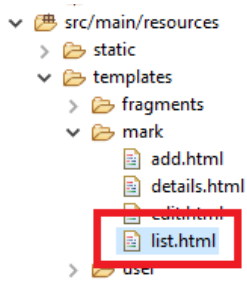
Comprobamos que el sistema de paginación funciona de la forma esperada.

Accedemos a la aplicación utilizando el perfil:

- DNI: 99999977E password: 123456 -> ROLE_PROFESSOR



Unicamente nos falta modificar las llamadas a **/mark/list/update** que realizamos desde el codigo JavaScript en dos puntos de la vista **/mark/list**



El primero el botón de actualizar que habíamos colocado encima de la tabla que mostraba las notas.

Debemos preparar el script para incluir parámetros thymeleaft (incluimos el **th:inline="javascript"** y el **CDATA)**.

Dentro del script para incluir a traves de thymeleaft a los parámetros de la URL usamos **[[${param.<nombre del parámetro>}]]** esta sentencia nos devuelve un **null** si no hay parámetro o un **array** de una única posición sí hay parámetro. Para acceder al valor del parámetro tenemos que consultar la posición [0] del array.

```
  <p>Las notas que actualmente figuran en el sistema son las siguientes:</p>
    <button type="button" id="updateButton" class="btn btn-default">Actualizar</button>
  <script th:inline="javascript">
        /*<![CDATA[*/
```

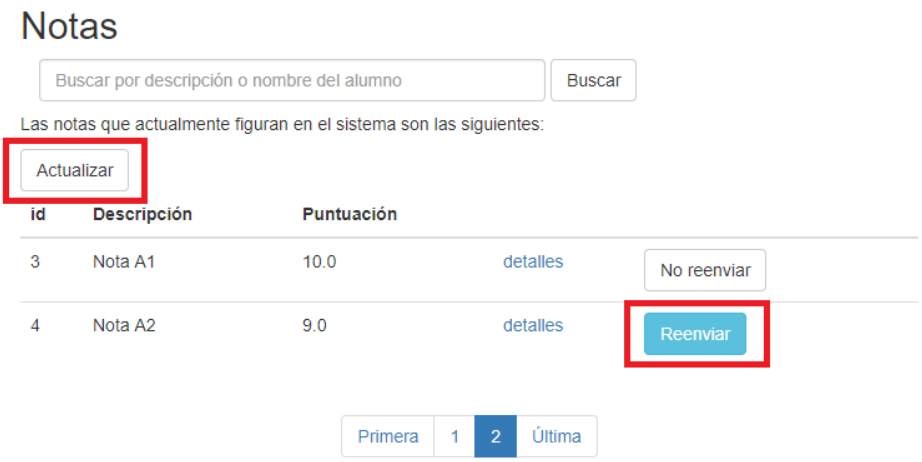
```
        $( "#updateButton" ).click(function() {
                var numberPage = [[${param.page}]];
                var urlUpdate = '/mark/list/update';
                if ( numberPage != null ){
                        urlUpdate += "?page="+numberPage[0];
                }
                $("#tableMarks").load(urlUpdate);
        });
        /*]]>*/

  </script>
```

La segunda invocación de **/mark/list/update** se encuentra en los botones de **Reenviar / No reenviar** la actividad que se mostraba solo a los usuarios con ROLE_STUDENT.

El procedimiento a seguir va a ser el mismo que el caso anterior, <u>sería adecuado sacar el código común a una función</u>

```
<div sec:authorize="hasRole('ROLE_STUDENT')" >
    <div th:if="${mark.resend}">

        <button type="button" th:id="${'resendButton' + mark.id}"
                class="btn btn-info">Reenviar</button>
        <script th:inline="javascript">
          /*<![CDATA[*/
          $( "#resendButton[[${mark.id}]]" ).click(function() {
              $.get( "/mark/[[${mark.id}]]/noresend", function( data ) {
                  var numberPage = [[${param.page}]];
                  var urlUpdate = '/mark/list/update';
                  if ( numberPage != null ){
                      urlUpdate += "?page="+numberPage[0];
                  }
                  $("#tableMarks").load(urlUpdate);
              });
          });
          /*]]>*/
        </script>
    </div>
    <div th:unless="${mark.resend}">

        <button type="button" th:id="${'noresendButton' + mark.id}"
                class="btn btn-default">No reenviar</button>
        <script th:inline="javascript">
          /*<![CDATA[*/
          $( "#noresendButton[[${mark.id}]]" ).click(function() {
              $.get( "/mark/[[${mark.id}]]/resend", function( data ) {
                  var numberPage = [[${param.page}]];
                  var urlUpdate = '/mark/list/update';
                  if ( numberPage != null ){
                      urlUpdate += "?page="+numberPage[0];
                  }
                  $("#tableMarks").load(urlUpdate);
              });
          });
          /*]]>*/
        </script>
    </div>
</div>
```

Guardamos los cambios y probamos la aplicación (para probar la paginación más fácilmente podemos modificar la configuración **CustomConfiguration** de forma que se muestren solo dos registros por página.



6.6 Ejercicio propuesto

Implementar paginación en la gestión de usuarios.