



# Sistemas Distribuidos e Internet

**Acceso a datos, Autenticación y control de acceso y Validación en el servidor**

**Sesión- 3**  
**Curso 2017/ 2018**



## Contenido

1	Thymeleaf – fragmentos.....	3
2	Internacionalización.....	4
2.1	Modificar la configuración de Sprint Boot.....	4
2.2	Definir las fuentes de los mensajes.....	5
2.3	Modificar las vistas.....	7
2.4	Ejercicio propuesto.....	10
3	Acceso a datos con entidades relacionadas.....	10
3.1	Modelos de datos.....	10
3.1.1	Entidad User.....	10
3.1.2	Entidad Mark.....	11
3.2	Repositorios JPA.....	13
3.3	Definir servicios.....	14
3.4	Definir y actualizar controladores.....	17
3.5	Definir y actualizar vistas.....	19
3.6	Ejercicio propuesto.....	23
4	Autenticación y control de acceso.....	24
4.1	Añadir las dependencias del proyecto.....	24
4.2	Definición del password.....	24
4.3	Repositorios.....	25
4.4	Servicios.....	25
4.4.1	Servicio UserDetailsService de Spring Security.....	25
4.4.2	Servicio SecurityService.....	27
4.4.3	Actualizar el servicio UserService.....	28
4.4.4	Actualizar el Controlador UsersController.....	29
4.4.5	Configuración del adaptador de seguridad.....	30
4.5	Definir y actualizar vistas.....	31
4.5.1	Vista signup.html (Solo para registrar nuevos ESTUDIANTES).....	31
4.5.2	Vista login.html (para identificar a cualquier usuario).....	32
4.5.3	Vista Home.html (Vista tras la identificación correcta).....	33
4.5.4	Acceso al usuario autenticado.....	34



5	Validación de datos en el servidor.....	35
5.1	Ejercicio propuesto .....	40

## 1 Thymeleaf – fragmentos

Thymeleaf ofrece la posibilidad de actualizar solo partes concretas de una vista mediante AJAX.

Por ejemplo, si quisiéramos incluir un mecanismo para actualizar únicamente la tabla de notas presente en **list.html** podríamos definir toda la tabla como un fragmento **th:fragment="tableMarks"**, también deberíamos darle una id para poder acceder a él de forma rápida desde jQuery.

```
<div class="container">
  <h2>Notas</h2>
  <p>Las notas que actualmente figuran en el sistema son las siguientes:</p>
  <div class="table-responsive">
    <table class="table table-hover" th:fragment="tableMarks" id="tableMarks">
      <thead>
        <tr>
          <th class="col-md-1">id</th>
```

Ahora nos hace falta definir una respuesta en **MarkControllers** para actualizar la lista de notas. Utilizaremos la url **/mark/list/update**, la única diferencia respecto a **list/** es que **no** retorna toda la vista sino solo el fragmento **tableMarks**

```
@RequestMapping("/mark/list/update")
public String updateList(Model model){
    model.addAttribute("markList", marksService.getMarks() );
    return "mark/list :: tableMarks";
}
```

Finalmente, solo tenemos que incluir un botón que relance la llamada a **/list/update** y sustituya el contenido de la tabla con id **tableMarks**. (En lugar de un botón también podríamos seguir otras estrategias: cada N segundos, o cada vez que la tabla reciba el foco, etc.)

Incluimos el botón en **list.html**, al pulsarlo se realizará una llamada con jQuery a la URL **/list/update** y sustituirá el antiguo contenido de la tabla por que acaba de obtener (función **load()**).

```
<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">
```

El nuevo botón actualizar recargará únicamente la tabla (probamos a crear una nueva nota desde otra pestaña y a actualizar la lista).



### Notas

Las notas que actualmente figuran en el sistema son las siguientes:

Actualizar					
id	Descripción	Puntuación			
1	Ejercicio22	5.0	<a href="#">detalles</a>	<a href="#">modificar</a>	<a href="#">eliminar</a>
2	Ejercicio2	10.0	<a href="#">detalles</a>	<a href="#">modificar</a>	<a href="#">eliminar</a>

## 2 Internacionalización

En este apartado veremos cómo podemos añadir internacionalización usando thymeleaf. Para configurar nuestra aplicación web para que soporte internacionalización, se deben seguir los siguientes pasos:

### 2.1 Modificar la configuración de Sprint Boot

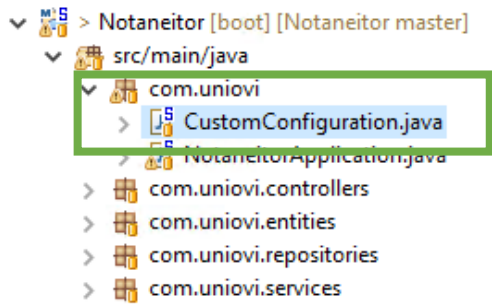
Debemos añadir un bean de tipo **LocaleResolver** y otro de tipo **LocaleChangeInterceptor**.

**LocaleResolver -> detectar localización.** Para que nuestra aplicación pueda determinar qué localización está utilizando actualmente, necesitamos agregar un bean **LocaleResolver**. La interfaz de **LocaleResolver** tiene implementaciones que determinan el entorno local actual basado en la sesión, las cookies, el encabezado Accept-Language o en un valor fijo. En nuestro caso iniciaremos con el idioma español por defecto ("ES").

**LocaleChangeInterceptor -> detectar parámetro de idioma.** Agregamos también un bean interceptor (**LocaleChangeInterceptor**) que nos permitirá utilizar el parámetro del idioma añadido a una petición. Finalmente, se debe añadir este bean al registro de interceptores de la aplicación, sobrescribiendo en método **addInterceptors**. Al interceptor le añadiremos un parámetro llamado **lang** que es el que utilizaremos en las peticiones.

Para que esto funcione debemos modificar la clase de configuración (**configuration**) de nuestra aplicación. Hasta ahora estamos utilizando la configuración por defecto. Ahora crearemos una nueva configuración CustomConfiguration que agregará los Beans para la gestión de idioma y sobrescribirá el método **addInterceptors**.

Comenzamos creando la clase **CustomConfiguration** en nuestro proyecto:



En la propia clase definiremos Beans **LocaleResolver**, **LocaleChangeInterceptor** y el método **addInterceptors**, como se muestra a continuación.

```
package com.uniovi;
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;

@Configuration
public class CustomConfiguration extends WebMvcConfigurerAdapter{

    @Bean
    public LocaleResolver localeResolver() {
        SessionLocaleResolver localeResolver = new SessionLocaleResolver();
        localeResolver.setDefaultLocale(new Locale("es", "ES"));
        return localeResolver;
    }

    @Bean
    public LocaleChangeInterceptor localeChangeInterceptor() {
        LocaleChangeInterceptor localeChangeInterceptor =
            new LocaleChangeInterceptor();
        localeChangeInterceptor.setParamName("lang");
        return localeChangeInterceptor;
    }

    @Override
    public void addInterceptors(InterceptorRegistry registry) {
        registry.addInterceptor(localeChangeInterceptor());
    }
}
```

## 2.2 Definir las fuentes de los mensajes

De forma predeterminada, la aplicación buscará archivos de mensajes que contengan las claves y valores de internacionalización en la carpeta **src/main/resources**.

El archivo de la configuración del idioma local predeterminada tendrá el nombre **messages.properties** y los archivos de cada idioma que soporte nuestra aplicación se

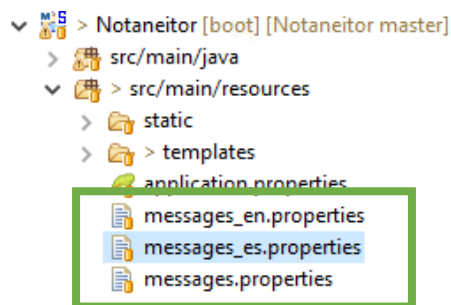


denominarán ***messages\_XX.properties***, donde XX es el código de configuración de idioma especificado, ej ES para español, EN para inglés, etc.

Las claves para los valores que se localizarán en estos archivos tienen que ser las mismas y con los valores apropiados para el idioma que corresponda.

Si no existe una clave en una configuración para un idioma específico, la aplicación volverá el valor de la configuración del idioma predeterminado.

En nuestro caso vamos a crear tres ficheros de configuración para que nuestra aplicación soporte dos idiomas: ***messages.properties*** (idioma por defecto), ***messages\_es.properties*** (Español) y ***messages\_en.properties*** (Inglés).



Ahora añadimos los mensajes de los diferentes idiomas en sus respectivos ficheros. En este caso vamos a añadir algunos mensajes simples. Como por ejemplo: un mensaje de bienvenida, un mensaje para cambiar el texto de los botones de login y de registrarse, etc.

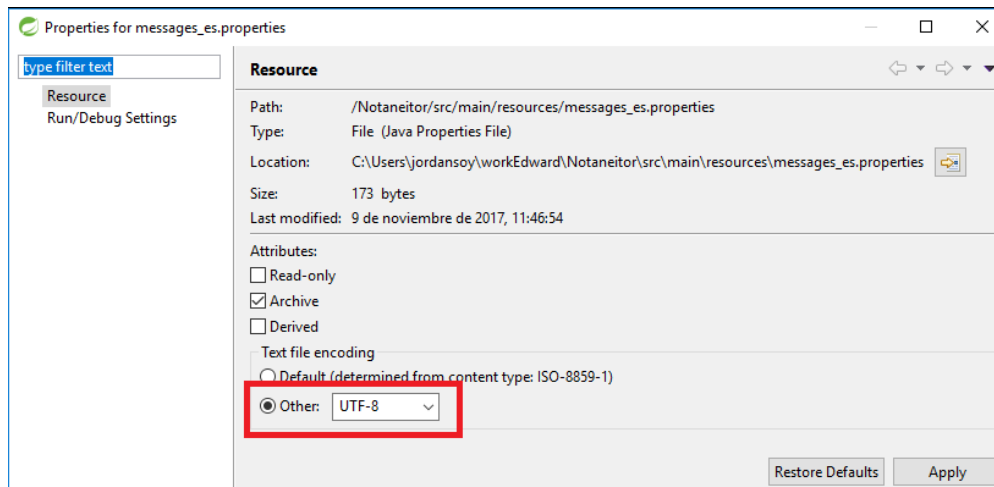
#### ***messages.properties y messages\_es.properties***

```
welcome.message=Bienvenidos a la página principal  
language.change= Idioma  
language.en=English  
language.es=Spanish  
login.message=Identificate  
signup.message=Registrate
```

#### ***messages\_en.properties***

```
welcome.message=Welcome to homepage  
language.change=Language  
language.en=English  
language.es=Spanish  
login.message=Login In  
signup.message=Sign Up
```

Es importante verificar que estos ficheros esten en formato **UTF-8** (podemos seleccionar el formato botón derecho sobre el fichero -> **Properties**).



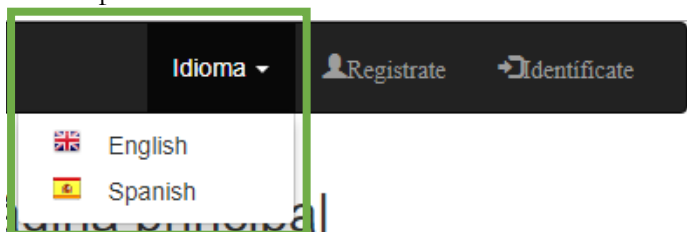
## 2.3 Modificar las vistas

En el motor *thymeleaf* accedemos a los valores de las variables usando las claves, con la sintaxis ***#{clave}***:

Modificamos la página *index.html* para internacionalizar el mensaje de bienvenida.

```
<nav th:replace="fragments/nav"/>
<div class="container" style="text-align: center">
  <h2 th:text="#{welcome.message}"></h2>
  <p>Lorem ipsum dolor sit amet, consectetur adipiscing</p>
</div>
<footer th:replace="fragments/footer"/>
```

Ahora vamos a modificar el **<nav>** de la aplicación para que el usuario pueda cambiar el idioma por defecto si lo desea.



Realizamos los siguientes cambios en **fragments/nav.html**:

- Añadiremos un dropdown (menú desplegable) para que el usuario pueda seleccionar el idioma. Como se muestra en la figura las opciones de los idiomas aparecen con la bandera del país a la izquierda del nombre, por esto debemos copiar estas dos imágenes dentro de la carpeta *img* del proyecto. **Estas imágenes están disponibles en los recursos en el campus.**



▼ > Notaneitor [boot] [Notaneitor master]

▼ > src/main/java

▼ > src/main/resources

▼ > static

▼ > CSS

▼ > img

if\_spain\_flag.png

if\_uk\_flag.png

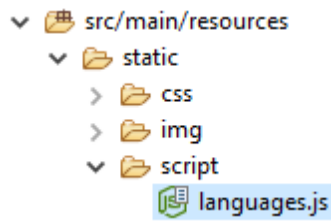
student-48.png

- En el fichero **fragmetns/nav.html**, modificamos los textos de algunas de las opciones de menú, para que cambien en función del idioma.

```
<div class="collapse navbar-collapse" id="myNavbar">
  <ul class="nav navbar-nav">
    <li class="active"><a href="/mark/list">Ver Notas</a></li>
    <li><a href="/mark/add">Agregar Nota</a></li>
    <li><a href="/mark/filter">Filtrar</a></li>
  </ul>
  <ul class="nav navbar-nav navbar-right">
    <li class="dropdown">
      <a id="btnLanguage" href="#"
        class="dropdown-toggle" data-toggle="dropdown" role="button"
        aria-haspopup="true" aria-expanded="false"> <span
          th:text="#{language.change}"></span> <span class="caret"></span>
      </a>
      <ul id="languageDropdownMenuButton" class="dropdown-menu">
        <li>
          <a id="btnEnglish" value="EN">  <span
              th:text="#{language.en}">Inglés</span>
          </a>
        </li>
        <li>
          <a id="btnSpanish" value="ES">  <span
              th:text="#{language.es}">Español</span>
          </a>
        </li>
      </ul>
    </li>
    <li>
      <a href="/signup" th:text="#{signup.message}">
        <span class="glyphicon glyphicon-user"></span>
        Registrare
      </a>
    </li>
    <li>
      <a href="/login" th:text="#{login.message}">
        <span class="glyphicon glyphicon-log-in"></span>
        Identificate
      </a>
    </li>
  </ul>
</div>
```

Por el momento el selector de idioma no está funcional, incluiremos un script para realizar el cambio de idioma. Vamos a crear un fichero **languages.js** en el directorio **static/script** que tendrá el código necesario para gestionar los idiomas soportados en la aplicación, utilizando **jQuery**.



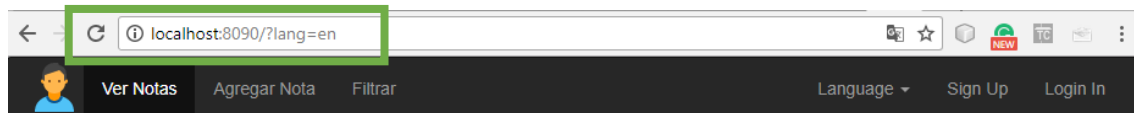


```
$(document).ready(function() {  
    $("#languageDropdownMenuButton a").click(function(e) {  
        e.preventDefault(); // cancel the link behaviour  
        var languageSelectedText = $(this).text();  
        var languageSelectedValue = $(this).attr("value");  
  
        $("#btnLanguage").text(languageSelectedText);  
        window.location.replace('?lang=' + languageSelectedValue);  
        return false;  
    });  
});
```

Finalmente, incluimos la referencia del fichero **languages.js** en la vista **/nav/head.html**, es importante incluirlo después de jQuery (ya que lo utilizamos dentro del script)

```
<head>  
  <meta charset="utf-8" />  
  <meta name="viewport" content="width=device-width, initial-scale=1" />  
  <script  
    src="https://ajax.googleapis.com/ajax/libs/jquery/3.2.0/jquery.min.js">  
  </script>  
  <script  
    src="https://maxcdn.bootstrapcdn.com/bootstrap/3.3.7/js/bootstrap.min.js">  
  </script>  
  <link rel="stylesheet"  
    href="https://maxcdn.bootstrapcdn.com/bootstrap/3.3.7/css/bootstrap.min.css" />  
  <link rel="stylesheet" href="/css/custom.css" />  
  <script src="/script/Languages.js"> </script>  
</head>
```

Cuando seleccionado un idioma u otro, se puede observar en la siguiente imagen que la URL de la aplicación se modifica dinámicamente, añadiendo un parámetro (**?lang=en**) a la petición indicado el idioma seleccionado.



Welcome to homepage

Lorem ipsum dolor sit amet, consectetur adipiscing



## 2.4 Ejercicio propuesto

Modificar las páginas actuales para que soporten internacionalización en todos sus apartados.

## 3 Acceso a datos con entidades relacionadas

Vamos a incluir **usuarios** con dos roles (roles: alumno y profesor). También se establecerá una relación entre **usuarios** y **notas**. Se definirá una relación de uno a muchos, es decir, un usuario podrá tener varias notas y una nota estará asociada a un usuario específico.

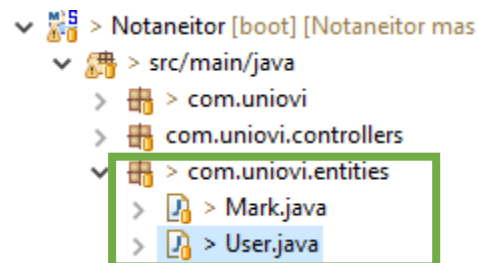
### 3.1 Modelos de datos

Para esta sección necesitaremos dos entidades para definir el modelo de datos, serán las entidades **User** y **Mark**.

#### 3.1.1 Entidad User

Creamos una nueva clase **User**, definimos relación de uno a muchos con la entidad **Mark** usando la notación **@OneToMany**.

Utilizamos el atributo **mappedBy** indica que la entidad **user** es la inversa de la relación y que será una relación en cascada **cascade = CascadeType.ALL**. Por ejemplo, si se borra un usuario se borrará en cascada las notas de ese usuario. **fetch = FetchType.EAGER** se puede utilizar en relaciones **OneToMany** o **ManyToMany** para indica que la estructura de datos se carga de forma proactiva (en lugar de perezosa) en el mismo momento que se carga la entidad User.



```
package com.uniovi.entities;

import javax.persistence.*;
import java.util.Set; //A collection that contains no duplicate elements

@Entity
public class User {

    @Id
    @GeneratedValue
    private long id;
    @Column(unique=true)
    private String dni;
    private String name;
    private String lastName;
    private String role;

    @OneToMany(mappedBy = "user", cascade = CascadeType.ALL)
```



```
private Set<Mark> marks;

public User(String dni, String name, String lastName) {
    super();
    this.dni = dni;
    this.name = name;
    this.lastName = lastName;
}

public User() {
}

public long getId() {
    return id;
}

public void setId(long id) {
    this.id = id;
}

public String getDni() {
    return dni;
}

public void setDni(String dni) {
    this.dni = dni;
}

public String getName() {
    return name;
}

public void setName(String name) {
    this.name = name;
}

public String getLastName() {
    return lastName;
}

public void setLastName(String lastName) {
    this.lastName = lastName;
}

public void setMarks(Set<Mark> marks) {
    this.marks = marks;
}

public Set<Mark> getMarks() {
    return marks;
}

public String getFullName() {
    return this.name + " " + this.lastName;
}
}
```

### 3.1.2 Entidad Mark

Modificamos la entidad **Mark** de la siguiente forma:

- Creamos la relación de mucho a uno con la entidad usuarios (User) con la anotación **@ManyToOne** y mediante la columna **user\_id** usando la notación



**@JoinColumn** que especifica la columna que va a crear una asociación entre las entidades.

- Creamos un nuevo constructor donde reciba un usuario
- Creamos métodos get y set para usuario

```
package com.uniovi.entities;

import javax.persistence.Entity;
import javax.persistence.GeneratedValue;
import javax.persistence.Id;
import javax.persistence.JoinColumn;
import javax.persistence.ManyToOne;

@Entity
public class Mark {
    @Id
    @GeneratedValue
    private Long id;
    private String description;
    private Double score;

    @ManyToOne
    @JoinColumn(name = "user_id")
    private User user;

    public Mark(Long id, String description, Double score) {
        super();
        this.id = id;
        this.description = description;
        this.score = score;
    }

    public Mark(String description, Double score, User user) {
        super();
        this.description = description;
        this.score = score;
        this.user = user;
    }

    public Mark() {
    }

    public Long getId() {
        return id;
    }

    public void setId(Long id) {
        this.id = id;
    }

    public String getDescription() {
        return description;
    }

    public void setDescription(String description) {
        this.description = description;
    }

    public Double getScore() {
        return score;
    }

    public void setScore(Double score) {
        this.score = score;
    }
}
```



```
public User getUser() {  
    return user;  
}  
  
public void setUser(User user) {  
    this.user = user;  
}  
  
}
```

A continuación, se resume en que consiste cada una de las anotaciones utilizadas:

**@Entity:** Especifica que la clase es una entidad. Esta anotación se aplica a la clase de entidad.

**@Id:** Especifica la clave primaria de una entidad.

**@Column** asigna el campo de la entidad a la columna específica. Si se omite **@Column**, se utiliza el valor predeterminado: el nombre de campo de la entidad.

**@OneToMany:** Define una asociación de muchos valores con multiplicidad de uno a muchos.

**@ManyToOne:** Define una asociación de valor único para otra clase de entidad que tiene multiplicidad de muchos a uno

**@JoinColumn:** Especifica una columna para unir una asociación de entidades o una colección de elementos. Si la anotación **JoinColumn** en sí es predeterminada, se supone una sola columna de unión y se aplican los valores predeterminados. **mappedBy** indica que la entidad es la inversa de la relación.

## 3.2 Repositorios JPA

Al igual que hicimos previamente para la entidad **Mark**, ahora debemos crear un repositorio **JPA** para la entidad **User**, la clase se llamará **UsersRepository**

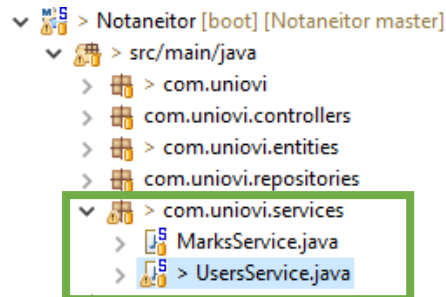
```
> Notaneitor [boot] [Notaneitor master]  
  > src/main/java  
    > com.uniovi  
      > com.uniovi.controllers  
      > com.uniovi.entities  
      > com.uniovi.repositories  
        > MarksRepository.java  
        > UsersRepository.java
```

```
package com.uniovi.repositories;  
  
import com.uniovi.entities.*;  
import org.springframework.data.repository.CrudRepository;  
  
public interface UsersRepository extends CrudRepository<User, Long>{}
```



### 3.3 Definir servicios

Creamos un nuevo servicio **UserService** para gestionar lo relativo a los usuarios, este servicio va a ser casi idéntico al utilizado por las notas.



```
package com.uniovi.services;

import java.util.*;
import javax.annotation.PostConstruct;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Service;
import com.uniovi.entities.Mark;
import com.uniovi.entities.User;
import com.uniovi.repositories.UsersRepository;

@Service
public class UserService {

    @Autowired
    private UsersRepository usersRepository;

    @PostConstruct
    public void init() {
    }

    public List<User> getUsers() {
        List<User> users = new ArrayList<User>();
        usersRepository.findAll().forEach(users::add);
        return users;
    }

    public User getUser(Long id) {
        return usersRepository.findOne(id);
    }

    public void addUser(User user) {
        usersRepository.save(user);
    }

    public void deleteUser(Long id) {
        usersRepository.delete(id);
    }
}
```

Como necesitaremos varios usuarios y notas para ver en funcionamiento la aplicación, vamos a crear un servicio de prueba **InsertSampleDataService**, utilizaremos el método **init()** de este servicio para crear dinámicamente varios usuarios con sus notas.

```
package com.uniovi.services;

import java.util.HashSet;
```



```
import java.util.Set;
import javax.annotation.PostConstruct;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Service;
import com.uniovi.entities.Mark;
import com.uniovi.entities.User;

@Service
public class InsertSampleDataService {

    @Autowired
    private UsersService usersService;

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

        Set user1Marks = new HashSet<Mark>() {
            {
                add(new Mark("Nota A1", 10.0, user1));
                add(new Mark("Nota A2", 9.0, user1));
                add(new Mark("Nota A3", 7.0, user1));
                add(new Mark("Nota A4", 6.5, user1));
            }
        };
        user1.setMarks(user1Marks);

        Set user2Marks = new HashSet<Mark>() {
            {
                add(new Mark("Nota B1", 5.0, user2));
                add(new Mark("Nota B2", 4.3, user2));
                add(new Mark("Nota B3", 8.0, user2));
                add(new Mark("Nota B4", 3.5, user2));
            }
        };
        user2.setMarks(user2Marks);

        Set user3Marks = new HashSet<Mark>() {
            {
                ;
                add(new Mark("Nota C1", 5.5, user3));
                add(new Mark("Nota C2", 6.6, user3));
                add(new Mark("Nota C3", 7.0, user3));
            }
        };
        user3.setMarks(user3Marks);

        Set user4Marks = new HashSet<Mark>() {
            {
                add(new Mark("Nota D1", 10.0, user4));
                add(new Mark("Nota D2", 8.0, user4));
                add(new Mark("Nota D3", 9.0, user4));
            }
        };
        user4.setMarks(user4Marks);
        usersService.addUser(user1);
        usersService.addUser(user2);
        usersService.addUser(user3);
        usersService.addUser(user4);
        usersService.addUser(user5);
    }
}
```



```
        usersService.addUser(user6);  
  
    }  
  
}
```

Modificamos la configuración de la aplicación para que elimine los datos anteriormente guardados en la base de datos y cree unos nuevos. Modificamos el fichero *application.properties* habilitando la creación de la base de datos.

```
# Crear modelo de nuevo  
spring.jpa.hibernate.ddl-auto=create
```

Sí quisieramos desactivar el servicio bastaría con eliminar la anotación etiqueta **@Service**

```
@Service  
public class InsertSampleDataService {
```





### 3.4 Definir y actualizar controladores

Comenzamos actualizando el controlador **MarksController**

- Para añadir y editar una nota se necesita tener la lista de usuarios de la aplicación. Esto se debe a que cuando se crea o se edita una nota debe estar asignada un usuario de la aplicación (se crean las relaciones **Usuario-> Notas**). Más adelante estableceremos un sistema de roles: estudiantes y profesores.

Vamos a crear una variable **usersList** con la lista de usuarios y se la vamos a enviar alas vistas, la vista permitirá por lo tanto seleccionar un usuario de la aplicación.

- En **POST /mark/edit/<id>** solamente vamos a permitir modificar el atributo **score** y **description**, no se podrá por lo tanto modificar el atributo usuario. Obtenemos nota original correspondiente a la **<id>**, sobrescribimos los atributos **score** y **description**, despues salvamos el objeto en el repositorio.

```
package com.uniovi.controllers;

import org.springframework.beans.factory.annotation.*;
import org.springframework.stereotype.Controller;
import org.springframework.ui.Model;
import org.springframework.web.bind.annotation.*;
import com.uniovi.entities.*;
import com.uniovi.services.MarksService;
import com.uniovi.services.UsersService;

@Controller
public class MarksController {

    @Autowired //Inyectar el servicio
    private MarksService marksService;

    @Autowired
    private UsersService userService;

    @RequestMapping("/mark/list/update")
    public String updateList(Model model){
        model.addAttribute("markList", marksService.getMarks() );
        return "mark/list :: tableMarks";
    }

    @RequestMapping("/mark/list")
    public String getList(Model model){
        model.addAttribute("markList", marksService.getMarks() );
        return "mark/list";
    }

    @RequestMapping(value="/mark/add", method=RequestMethod.POST )
    public String setMark(@ModelAttribute Mark mark){
        marksService.addMark(mark);
        return "redirect:/mark/list";
    }

    @RequestMapping("/mark/details/{id}" )
    public String getDetail(Model model, @PathVariable Long id){
        model.addAttribute("mark", marksService.getMark(id));
        return "mark/details";
    }

    @RequestMapping("/mark/delete/{id}" )
    public String deleteMark(@PathVariable Long id){
```



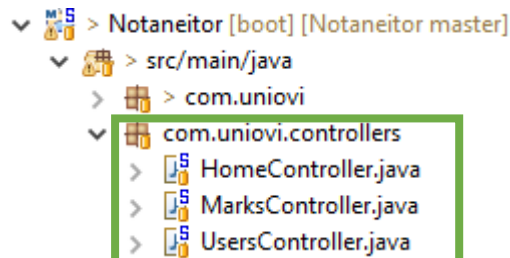
```
marksService.deleteMark(id);
return "redirect:/mark/list";
}

@RequestMapping(value="/mark/add")
public String getMark(Model model){
    model.addAttribute("usersList", userService getUsers());
    return "mark/add";
}

@RequestMapping(value="/mark/edit/{id}")
public String getEdit(Model model, @PathVariable Long id){
    model.addAttribute("mark", marksService.getMark(id));
    model.addAttribute("usersList", userService getUsers());
    return "mark/edit";
}

@RequestMapping(value="/mark/edit/{id}", method=RequestMethod.POST)
public String setEdit(Model model, @PathVariable Long id, @ModelAttribute Mark mark){
    Mark original = marksService.getMark(id);
    // modificar solo score y description
    original.setScore(mark.getScore());
    original.setDescription(mark.getDescription());
    marksService.addMark(original);
    return "redirect:/mark/details/"+id;
}
}
```

Definimos el nuevo controlador **UsersController**.



Será prácticamente igual al controlador de las **notas**, pero basado en los **usuarios**, **permitirá**: listar, añadir, ver detalles, modificar y eliminar.

```
package com.uniovi.controllers;

import org.springframework.beans.factory.annotation.*;
import org.springframework.stereotype.Controller;
import org.springframework.ui.Model;
import org.springframework.web.bind.annotation.*;
import com.uniovi.entities.*;
import com.uniovi.services.UsersService;

@Controller
public class UsersController {

    @Autowired
    private UsersService userService;

    @RequestMapping("/user/list" )
    public String getListado(Model model){
```



```
        model.addAttribute("userslist", userService getUsers());
        return "user/list";
    }

    @RequestMapping(value="/user/add")
    public String getUser(Model model){
        model.addAttribute("userslist", userService getUsers());
        return "user/add";
    }

    @RequestMapping(value="/user/add", method=RequestMethod.POST )
    public String setUser(@ModelAttribute User user){
        userService.addUser(user);
        return "redirect:/user/list";
    }

    @RequestMapping("/user/details/{id}" )
    public String getDetail(Model model, @PathVariable Long id){
        model.addAttribute("user", userService.getUser(id));
        return "user/details";
    }

    @RequestMapping("/user/delete/{id}" )
    public String delete(@PathVariable Long id){
        userService.deleteUser(id);
        return "redirect:/user/list";
    }

    @RequestMapping(value="/user/edit/{id}")
    public String getEdit(Model model, @PathVariable Long id){
        User user = userService.getUser(id);
        model.addAttribute("user", user);
        return "user/edit";
    }

    @RequestMapping(value="/user/edit/{id}", method=RequestMethod.POST)
    public String setEdit(Model model, @PathVariable Long id, @ModelAttribute User user){
        user.setId(id);
        userService.addUser(user);
        return "redirect:/user/details/"+id;
    }
}
```

### 3.5 Definir y actualizar vistas

Finalmente vamos a modificar las vistas para gestionar las **notas** y los **usuarios**.

Comenzamos modificando el fragmento del menú de navegación **fragments/nav.html** . Debemos actualizar las Url de los antiguos enlaces a **notas** y añadir la gestión de **alumnos**.

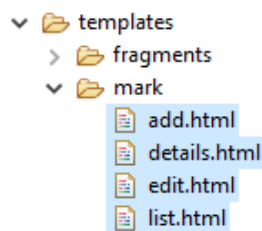
Vamos a agrupar las opciones en dos dropdown menú en **nav.html**.

```
<div class="collapse navbar-collapse" id="myNavbar">
    <ul class="nav navbar-nav">
        <li class="active"><a href="/list">Ver Notas</a></li>
        <li><a href="/add">Agregar Nota</a></li>
        <li><a href="/filter">Filtrar</a></li>
        <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><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">
  <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>
</ul>
```

Continuamos modificando el contenido de las antiguas vistas asociadas a **nota**, en la carpeta **templates/mark**.



**add.html**, agregaremos el sistema de selección de usuario.

Selección de usuario: Comprobamos si la vista recibe la lista **usersList** con todos los usuarios de la aplicación. Si obtenemos la **usersList** creamos un campo de selección **<select>** y un elemento **<option>** por cada usuario. El valor del campo **<option>** debe ser la clave primaria del usuario, **user.id**, el texto puede ser cualquier cadena que consideremos descriptiva.

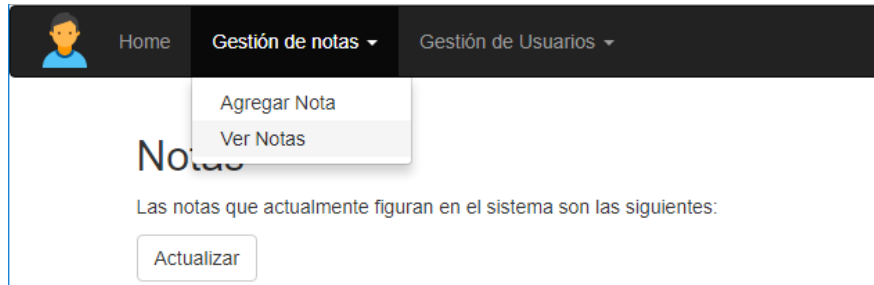
```
<form class="form-horizontal" method="post" action="/mark/add">
<div class="form-group">
  <label class="control-label col-sm-2" for="user">Alumno:</label>
  <div class="col-sm-10" th:if="{usersList != null and not #lists.isEmpty(usersList)}">
    <select id="user" class="form-control" name="user">
      <option th:each="user : ${usersList}">
        th:value="${user.id}"
        th:text="${user.dni}+ ' - '+${user.name}+ ' ' +${user.lastName}">
          Usuario
      </option>
    </select>
  </div>
</div>
</div>
```

**details.html**, agregaremos el atributo **mark.dni.user**

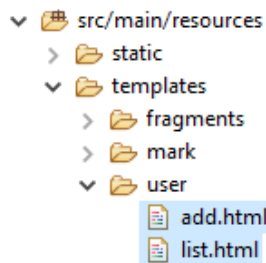
```
<div class="container">
  <h2>Detalles de la nota</h2>
  <div class="panel panel-default">
    <div class="panel-heading">DNI</div>
    <div class="panel-body" th:text="${mark.user.dni}">
      28276
    </div>
  </div>
  <div class="panel panel-default">
```



Llegamos a este punto ya podemos probar las funcionalidades dependientes de Nota.



Creamos la carpeta **/templates/user** en esta carpeta almacenaremos las vistas relacionadas con la entidad **User**. Comenzamos creando las vistas **add.html** y **list.html**



**user/add.html** nos permitirá incluir un usuario.

```
<!DOCTYPE html>
<html lang="en">
<head th:replace="fragments/head"/>
<body>

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

<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="dni">DNI:</label>
      <div class="col-sm-10">
        <input type="text" class="form-control" name="dni"
          placeholder="99999999Y" required="true" />
      </div>
    </div>
    <div class="form-group">
      <label class="control-label col-sm-2" for="name">Nombre:</label>
      <div class="col-sm-10">
        <input type="text" class="form-control" name="name"
          placeholder="Ejemplo: Juan" required="true" />
      </div>
    </div>
    <div class="form-group">
      <label class="control-label col-sm-2" for="lastName">Apellidos:</label>
      <div class="col-sm-10">
        <input type="text" class="form-control" name="lastName"
          placeholder="Ejemplo: Pérez Almonte" required="true" />
      </div>
    </div>
    <div class="form-group">
      <div class="col-sm-offset-2 col-sm-10">
        <button type="submit" class="btn btn-primary">Enviar</button>
      </div>
    </div>
  </form>
</div>
```



```
</div>
</form>
</div>

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

</body>
</html>
```

**users/list.html** , recorrerá la variable enviada por el controlador y listará todos los usuarios.

```
<!DOCTYPE html>
<html lang="en">
<head th:replace="fragments/head"/>
<body>

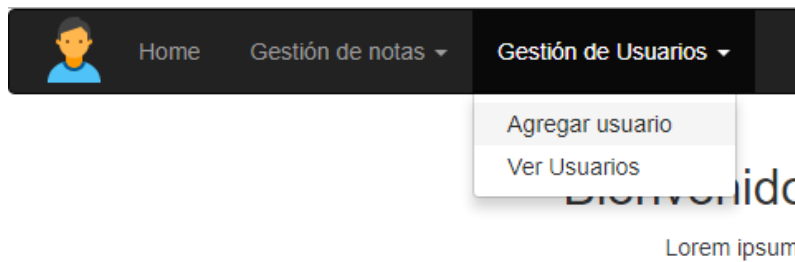
<!-- Barra de Navegación superior -->
<nav th:replace="fragments/nav"/>

<div class="container">
  <h2>Usuarios</h2>
  <p>Los usuarios que actualmente figuran en el sistema son los siguientes:</p>
  <div class="table-responsive">
    <table class="table table-hover">
      <thead>
        <tr>
          <th>DNI</th>
          <th>Nombre </th>
          <th>Apellidos </th>
          <th class="col-md-1"></th>
          <th class="col-md-1"></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="user : ${usersList}">
          <td th:text="${user.dni}">71888888X</td>
          <td th:text="${user.name}">Nombre del alumno</td>
          <td th:text="${user.lastName}">Apellidos del alumno</td>
          <td><a th:href="'/user/details/' + user.id">detalles</a></td>
          <td><a th:href="'/user/edit/' + user.id">modificar</a></td>
          <td><a th:href="'/user/delete/' + user.id">eliminar</a></td>
        </tr>
      </tbody>
    </table>
  </div>
</div>

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

</body>
</html>
```

Comprobamos que la aplicación nos permita **listar, crear y eliminar** usuarios correctamente.



**Importante:** Una vez ejecutada la aplicación podemos eliminar el `init()` presente en `UserServices` que se encargaba de incluir datos de prueba y cambiar la propiedad de `application.properties` a `spring.jpa.hibernate.ddl-auto=validate`. Sí no hacemos este cambio la base de datos se “reiniciará” con cada ejecución.

**Importante:** si en algún momento aparecen problemas para crear la base de datos borrar su esquema manualmente (podemos hacerlo mediante comandos o volviendo a extraer el contenido de `hsqldb.zip` )

### 3.6 Ejercicio propuesto

Completar el `editar` y `detalles` de usuario.



## 4 Autenticación y control de acceso

La autenticación y control de acceso a recursos en aplicaciones web puede ser un punto crítico en muchas aplicaciones. En esta sección veremos como hacer un sistema de identificación y control de acceso utilizando **Spring Boot** y **Spring Security**.

### 4.1 Añadir las dependencias del proyecto

La primera dependencia que vamos a incluir en el fichero **pom.xml** será **spring-boot-starter-security**, spring security que es un framework que centra en proporcionar autenticación y autorización para aplicaciones web de una manera comprensible y extensible.

Añadir la dependencia **thymeleaf-extras-springsecurity4**, se trata de un módulo extra, que no es parte de core Thymeleaf, proporciona un conjunto de elementos para trabajar con los objetos de spring security de una manera sencilla desde las plantillas Thymeleaf.

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

<dependency>
  <groupId>org.thymeleaf.extras</groupId>
  <artifactId>thymeleaf-extras-springsecurity4</artifactId>
</dependency>
```

Una vez añadidas estas dependencias Spring Security ya esta siendo utilizado en la aplicación.

### 4.2 Definición del password

Vamos modificar ahora la entidad **User**,

- Añadimos nuevos campos: **password** y **passwordConfirm** los necesitaremos para la creación de usuarios y para la autenticación.

```
@Entity
@Table(name = "user")
public class User {

    @Id
    @GeneratedValue
    private long id;
    @Column(unique=true)
    private String dni;
    private String name;
    private String lastName;

    private String password;
    @Transient //propiedad que no se almacena en la tabla.
    private String passwordConfirm;
```

Agregamos los **get** y **set** correspondientes.





```
public String getPassword() {  
    return password;  
}  
  
public void setPassword(String password) {  
    this.password = password;  
}  
  
public String getPasswordConfirm() {  
    return passwordConfirm;  
}  
  
public void setPasswordConfirm(String passwordConfirm) {  
    this.passwordConfirm = passwordConfirm;  
}
```

## 4.3 Repositorios

Modificamos ahora el **UsersRepository**

Como vamos a utilizar el **DNI** para identificar a los usuarios en el login definiremos un método **findByDni()**. el **DNI** será el nombre de usuario en esta aplicación. Las funciones **findBy<nombre atributo>** se dotan de funcionalidad de forma automática.

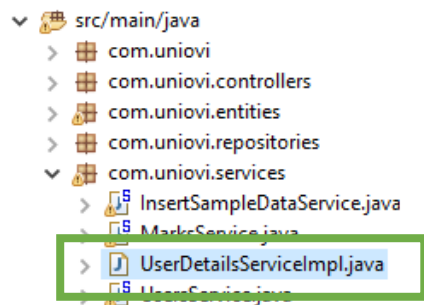
```
public interface UsersRepository extends CrudRepository<User, Long>{  
    User findByDni(String dni);  
}
```

## 4.4 Servicios

### 4.4.1 Servicio UserDetailsService de Spring Security

Para poder utilizar el sistema de autenticación y control de acceso con Spring Security, debemos implementar un servicio basado en **UserDetailsService**. **UserDetailsService** es una interfaz de **springframework.security.core** que ofrece una serie de métodos que nos permitirán gestionar la autenticación y el acceso.

Creamos el servicio **UserDetailsServiceImpl** y hacemos que implemente la interfaz **UserDetailsService**. En esta clase se define el método **loadUserByUsername(String name)**, en la implementación de esta función debemos devolver un **user (security.core.userdetails.User)** y sus roles, en nuestro caso el Username es de el dni del usuario (aunque podríamos haber utilizado otro valor unico, la id, el UO, etc.)



La clase **userdetails.User** de spring security implementa el interfaz **userdetails**, es la clase que utiliza el sistema de autenticación, no es el mismo **user** que el que maneja nuestra aplicación.

Spring security llama a las propiedades de identificación: **Username** y **password**.

Obtenemos el usuario de nuestra aplicación y creamos un **userdetails** con su DNI y password.

```
package com.uniovi.services;

import com.uniovi.entities.User;
import com.uniovi.repositories.UsersRepository;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.security.core.GrantedAuthority;
import org.springframework.security.core.authority.SimpleGrantedAuthority;
import org.springframework.security.core.userdetails.UserDetails;
import org.springframework.security.core.userdetails.UserDetailsService;
import org.springframework.security.core.userdetails.UsernameNotFoundException;
import org.springframework.stereotype.Service;
import java.util.*;

@Service("userDetailsService")
public class UserDetailsServiceImpl implements UserDetailsService{
    @Autowired
    private UsersRepository usersRepository;

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

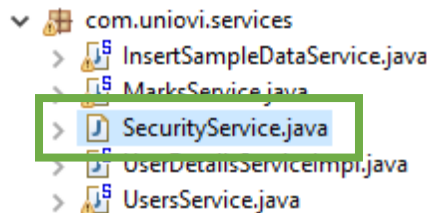
        Set<GrantedAuthority> grantedAuthorities = new HashSet<>();
        grantedAuthorities.add(new SimpleGrantedAuthority("ROLE_ESTUDIANTE"));

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



#### 4.4.2 Servicio SecurityService

Dentro del paquete *com.uniovi.services*, vamos a crear la clase **SecurityService** que tendrá dos métodos: *findLoggedInDni()* devolvera el usuario actual autenticado y el método *autoLogin()* que permitirá el inicio automatico de sesión después de que un usuario cree una cuenta. Este servicio se va a encargar de la autenticación de los usuarios.



```
package com.uniovi.services;

import org.slf4j.Logger;
import org.slf4j.LoggerFactory;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.security.authentication.AuthenticationManager;
import org.springframework.security.authentication.UsernamePasswordAuthenticationToken;
import org.springframework.security.core.context.SecurityContextHolder;
import org.springframework.security.core.userdetails.UserDetails;
import org.springframework.security.core.userdetails.UserDetailsService;
import org.springframework.stereotype.Service;

@Service
public class SecurityService {
    @Autowired
    private AuthenticationManager authenticationManager;

    @Autowired
    private UserDetailsService userDetailsService;

    private static final Logger logger = LoggerFactory.getLogger(SecurityService.class);

    public String findLoggedInDni() {
        Object userDetails = SecurityContextHolder.getContext().getAuthentication().getDetails();
        if (userDetails instanceof UserDetails) {
            return ((UserDetails)userDetails).getUsername();
        }

        return null;
    }

    public void autoLogin(String dni, String password) {
        UserDetails userDetails = userDetailsService.loadUserByUsername(dni);

        UsernamePasswordAuthenticationToken aToken = new UsernamePasswordAuthenticationToken(
            userDetails, password, userDetails.getAuthorities());

        authenticationManager.authenticate(aToken);

        if (aToken.isAuthenticated()) {
            SecurityContextHolder.getContext().setAuthentication(aToken);
            logger.debug(String.format("Auto login %s successfully!", dni));
        }
    }
}
```



#### 4.4.3 Actualizar el servicio UserService

Ahora debemos modificar **UserService** que es el servicio que gestiona la lógica de negocio de los **usuarios**, debemos incluir el nuevo atributo **password**,

Por seguridad, y para evitar robo de autenticación y inicio de sesión el **password** se debería almacenar cifrado.

Modificamos el método **addUser()** para que al guardar un objeto usuario **cifre el password**.

Creamos el nuevo método **findByDni (String dni)** para poder buscar usuario por su username: el **dni**.

```
@Service
public class UserService {

    @Autowired
    private UsersRepository usersRepository;

    @Autowired
    private BCryptPasswordEncoder bCryptPasswordEncoder;

    public List<User> getUsers() {
        List<User> users = new ArrayList<User>();
        usersRepository.findAll().forEach(users::add);
        return users;
    }

    public User getUser(Long id) {
        return usersRepository.findOne(id);
    }

    public void addUser(User user) {
        user.setPassword(bCryptPasswordEncoder.encode(user.getPassword()));
        usersRepository.save(user);
    }

    public User getUserByDni(String dni) {
        return usersRepository.findByDni(dni);
    }

    public void deleteUser(Long id) {
        usersRepository.delete(id);
    }
}
```

Como se ha modificado la entidad **User** vamos a volver a utilizar el servicio **InsertSampleDataService** para incluir nuevos datos de pruebas, usuarios creados con **password**.

Simplemente agregamos passwords a los datos de ejemplo anteriores.

```
@PostConstruct
public void init() {
    User user1 = new User("99999990A", "Pedro", "Díaz");
    user1.setPassword("123456");
    User user2 = new User("99999991B", "Lucas", "Núñez");
    user2.setPassword("123456");
}
```



```
User user3 = new User("99999992C", "María", "Rodríguez");
user3.setPassword("123456");
User user4 = new User("99999993D", "Marta", "Almonte");
user4.setPassword("123456");
User user5 = new User("99999997E", "Pelayo", "Valdes");
user5.setPassword("123456");
User user6 = new User("99999988F", "Edward", "Núñez");
user6.setPassword("123456");
```

Recordamos que para que los datos sean sustituidos en la base de datos el **application.properties** debe definir la siguiente propiedad:

```
spring.jpa.hibernate.ddl-auto=create
```

#### 4.4.4 Actualizar el Controlador UsersController

Primero inyectamos el servicio **SecurityService**.

Creamos los métodos **signup()** y **login()** para la registro y autenticación de usuarios, respectivamente. Hay que destacar, que en el método **signup()**, asignará automáticamente el role de alumno a usuario que crea una cuenta en la aplicación.

Finalmente implementamos el método **home()**, que redirija al usuario cuando la autenticación sea válida.

```
@Controller
public class UsersController {

    @Autowired
    private UsersService usersService;

    @Autowired
    private SecurityService securityService;
```

Incluimos la implementación de las funciones: **signup**, **login** y **home**.

**/signup** crea un nuevo usuario con el role STUDENT, lo identifica automáticamente y redirige la navegación a home.

**login** y **home** muestran únicamente sus vistas correspondientes.

```
@RequestMapping(value = "/signup", method = RequestMethod.GET)
public String signup() {
    return "signup";
}

@RequestMapping(value = "/signup", method = RequestMethod.POST)
public String signup(@ModelAttribute("user") User user, Model model) {
    usersService.addUser(user);
    securityService.autoLogin(user.getDni(), user.getPasswordConfirm());
    return "redirect:home";
}

@RequestMapping(value = "/login", method = RequestMethod.GET)
public String login(Model model) {
    return "login";
}

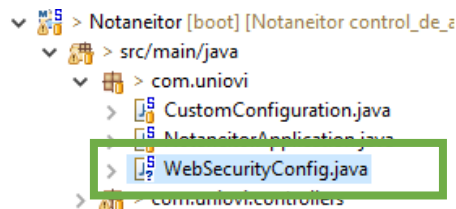
@RequestMapping(value = { "/home" }, method = RequestMethod.GET)
public String home(Model model) {
    return "home";
}
```



#### 4.4.5 Configuración del adaptador de seguridad

Crearemos una nueva configuración **WebSecurityConfig** en el paquete **com.uniovi**.

La clase **WebSecurityConfig** heredará de **WebSecurityConfigurerAdapter** e incluirá la anotación **@EnableWebSecurity**.



El adaptador nos permite especificar que recursos de la aplicación son, cuales no y bajo que roles se accede a los mismos

```
package com.uniovi;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import org.springframework.context.annotation.EnableWebSecurity;
import org.springframework.security.config.annotation.authentication.builders.AuthenticationManagerBuilder;
import org.springframework.security.config.annotation.web.builders.HttpSecurity;
import org.springframework.security.config.annotation.web.configuration.EnableWebSecurity;
import org.springframework.security.config.annotation.web.configuration.WebSecurityConfigurerAdapter;
import org.springframework.security.core.userdetails.UserDetailsService;
import org.springframework.security.crypto.bcrypt.BCryptPasswordEncoder;

@Configuration
@EnableWebSecurity
public class WebSecurityConfig extends WebSecurityConfigurerAdapter {

    @Autowired
    private UserDetailsService userDetailsService;

    @Bean
    public BCryptPasswordEncoder bCryptPasswordEncoder() {
        return new BCryptPasswordEncoder();
    }

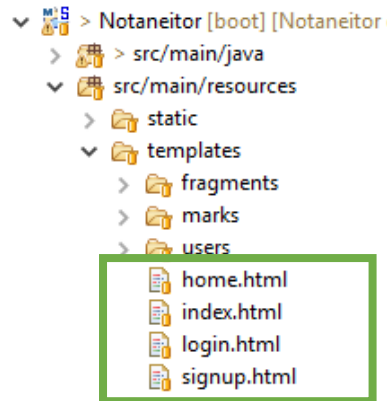
    @Override
    protected void configure(HttpSecurity http) throws Exception {
        http
            .csrf().disable()
            .authorizeRequests()
                .antMatchers("/css/**", "/img/**", "/script/**", "/", "/signup", "/login/**").permitAll()
                .anyRequest().authenticated()
                .and()
            .formLogin()
                .loginPage("/login")
                .permitAll()
                .defaultSuccessUrl("/home")
                .and()
            .logout()
                .permitAll();
    }

    @Autowired
    public void configureGlobal(AuthenticationManagerBuilder auth) throws Exception {
        auth.userDetailsService(userDetailsService).passwordEncoder(bCryptPasswordEncoder());
    }
}
```



## 4.5 Definir y actualizar vistas

Vamos a crear las siguientes vistas: *home.html*, *login.html* y *signup.html*



### 4.5.1 Vista signup.html (Solo para registrar nuevos ESTUDIANTES)

Creemos la página **signup.html** para que un usuario pueda registrarse. Realmente es la misma vista que **user/add.html**, añadiéndole los campos relativos al password:

```
<!DOCTYPE html>
<html lang="en">
<head th:replace="fragments/head"/>
<body>

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

<div class="container">

<h2>Regístrate como usuario</h2>
<form class="form-horizontal" method="post" action="/signup">
  <div class="form-group">
    <label class="control-label col-sm-2" for="dni">DNI:</label>
    <div class="col-sm-10">
      <input type="text" class="form-control" name="dni"
        placeholder="99999999Y" required="true" />
    </div>
  </div>
  <div class="form-group">
    <label class="control-label col-sm-2" for="name">Nombre:</label>
    <div class="col-sm-10">
      <input type="text" class="form-control" name="name"
        placeholder="Ejemplo: Juan" required="true" />
    </div>
  </div>
  <div class="form-group">
    <label class="control-label col-sm-2" for="lastName">Apellidos:</label>
    <div class="col-sm-10">
      <input type="text" class="form-control" name="lastName"
        placeholder="Ejemplo: Pérez Almonte" required="true" />
    </div>
  </div>
  <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="Entre el Password" />
    </div>
  </div>
</form>
</div>
```



```
<label class="control-label col-sm-2" for="passwordConfirm">Repita  
el Password:</label>  
<div class="col-sm-10">  
  <input type="password" class="form-control" name="passwordConfirm"  
    placeholder="Repita el Password" />  
</div>  
</div>  
<div class="form-group">  
  <div class="col-sm-offset-2 col-sm-10">  
    <button type="submit" class="btn btn-primary">Enviar</button>  
  </div>  
</div>  
</form>  
</div>  
<footer th:replace="fragments/footer"/>  
  
</body>  
</html>
```

#### 4.5.2 Vista login.html (para identificar a cualquier usuario)

Creamos la vista **login.html** para la autenticación de los usuarios.

Es importante que los atributos se llamen: **username** y **password** y que el formulario se dirija a **POST /login** (si nos fijamos no hemos implementado la respuesta **POST /login** en el controlador, esto es parte del SecurityService de Spring Security).

**Cross-Site Request Forgery (CSRF):** es un tipo de ataque que ocurre cuando un sitio web, correo electrónico, blog, mensaje instantáneo o programa malicioso hace que el navegador web de un usuario realice una acción no deseada en un sitio confiable para el cual el usuario está actualmente autenticado.

```
<!DOCTYPE html>  
<html lang="en">  
<head th:replace="fragments/head"/>  
<body>  
  
<nav th:replace="fragments/nav"/>  
  
<div class="container">  
<h2>Identificate</h2>  
<form class="form-horizontal" method="post" action="/login">  
  <div class="form-group">  
    <label class="control-label col-sm-2" for="username">DNI:</label>  
    <div class="col-sm-10">  
      <input type="text" class="form-control" name="username"  
        placeholder="Ejemplo: profSDI" required="true" />  
    </div>  
  </div>  
  <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="Entre el Password" required="true" />  
    </div>  
  </div>  
  <div class="form-group">  
    <div class="col-sm-offset-2 col-sm-10">  
      <button type="submit" class="btn btn-primary">Login</button>  
    </div>  
  </div>  
  <input type="hidden" name="{_csrf.parameterName}" value="{_csrf.token}" />  
</form>  
  
</div>
```





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

</body>
</html>
```

### 4.5.3 Vista Home.html (Vista tras la identificación correcta)

Finalmente creamos la página **home.html**, será la primera página que visualicen los usuarios logueados, **en este punto, tenemos un problema de seguridad ya que todos los usuarios tienen acceso a gestionar usuarios, notas, ... esto se resolverá más adelante.**

```
<!DOCTYPE html>
<html lang="en">
<head th:replace="fragments/head"/>
<body>

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

<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 :
    <b th:inline="text"> [[#{httpServletRequest.remoteUser}]] </b>
  </p>
</div>

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

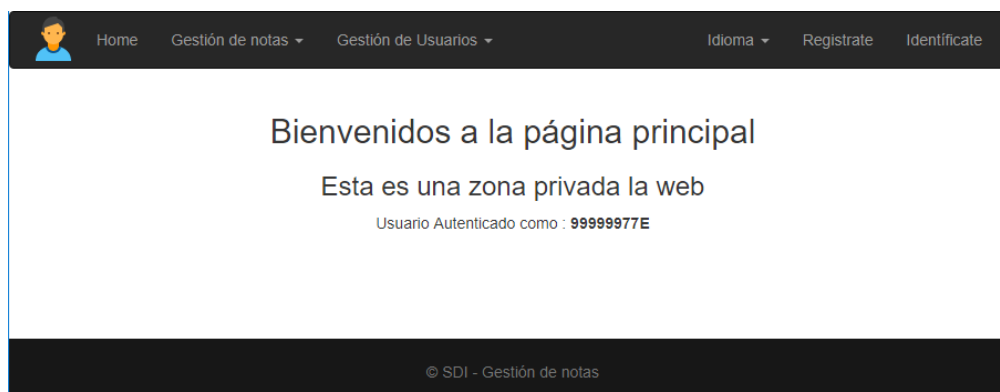
</body>
</html>
```

Ejecutamos la aplicación y probamos el sistema de identificación y autenticación, aquí tenemos algunos de los datos de prueba:

DNI: 99999990A      password: 123456

DNI: 99999988F      password: 123456

DNI: 99999977E      password: 123456





#### 4.5.4 Acceso al usuario autenticado

Vamos a realizar una pequeña modificación, haciendo que la vista **/home** muestre las notas del usuario autenticado.

Modificamos la vista **home** haciendo que procese un objeto **markList** el cual contendrá una lista de las notas asociadas al usuario, (recorremos este objeto de la misma forma que en la vista mark/list(

```
<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 th:inline="text"> [[#{HttpServletRequest.remoteUser}]] </span>
  </p>
  <p>
    Notas del 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 : ${markList}">
          <td th:text="${mark.id}"> 1</td>
          <td th:text="${mark.description}"> Ejercicio 1</td>
          <td th:text="${mark.score}">10</td>
          <td><a th:href="'${mark.details}' + mark.id">detalles</a></td>
          <td><a th:href="'${mark.edit}' + mark.id">modificar</a></td>
          <td><a th:href="'${mark.delete}' + mark.id">eliminar</a></td>
        </tr>
      </tbody>
    </table>
  </div>
</div>
```

Accedemos al controlador **UserController** y modificamos la respuesta a **GET /home**,

- Obtemos el objeto **Authentication** que almacena toda la información del usuario autenticado, con **getName** obtenemos el username (en este caso es el DNI)
- Utilizando el **userService** obtenemos toda la información del usuario autenticado.
- Una vez tenemos al usuario obtenemos su conjunto de calificaciones y las guardamos en el atributo **markList** para enviárselas a la vista.

\*Utilizando el objeto **Authentication** podemos obtener mucha información del usuario autenticado, su username, password (**getCredentials()**), roles (**getAuthorities()**) y otros detalles (**getDetails()**).



```
@RequestMapping(value = { "/home" }, method = RequestMethod.GET)
public String home(Model model) {
    Authentication auth = SecurityContextHolder.getContext().getAuthentication();
    String dni = auth.getName();
    User activeUser = userService.getUserByDni(dni);
    model.addAttribute("markList", activeUser.getMarks());
    return "home";
}
```

Sí ejecutamos la aplicación y nos autenticamos como DNI: 99999990A password: 123456 veremos que la vista **/home** nos muestre las notas de ese usuario.



## Bienvenidos a la página principal

Esta es una zona privada la web

Usuario Autenticado como : 99999990A

Notas del usuario

id	Descripción	Puntuación			
1	Nota A2	9.0	<a href="#">detalles</a>	<a href="#">modificar</a>	<a href="#">eliminar</a>
4	Nota A1	10.0	<a href="#">detalles</a>	<a href="#">modificar</a>	<a href="#">eliminar</a>
3	Nota A3	7.0	<a href="#">detalles</a>	<a href="#">modificar</a>	<a href="#">eliminar</a>
2	Nota A4	6.5	<a href="#">detalles</a>	<a href="#">modificar</a>	<a href="#">eliminar</a>

## 5 Validación de datos en el servidor

Cuando desarrollamos una aplicación web, es muy importante validar los datos de entrada que son enviados por los usuarios del sistema, para evitar errores y potenciales fallos de seguridad.

Las validaciones pueden realizarse en cliente mediante un lenguaje de script (ej., Javascript) o en el servidor. **Para evitar problema de seguridad lo recomendable es validar en el cliente y en el servidor.**

Para proporcionar validación de datos de entrada de usuarios podemos utilizar la interfaz **Validator** de Spring, la cual nos permite validar objetos de forma ágil. La interfaz **Validator** funciona utilizando un objeto **Errors** para informar de los errores ocurridos.

Dentro del paquete **com.uniovi.validators**, vamos a crear la clase **SignUpFormValidator** que implemente la interfaz **org.springframework.validation**, el primer formulario que vamos a validar es el formulario de registro de un nuevo usuario presente en **signup.html**

**DNI existente:** vamos a necesitar una función que nos permita buscar un usuario por Dni, para ver si este está duplicado. Accedemos a **UsersRepository** y agregamos la función

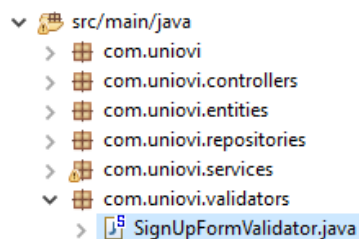


`findByDni` (no tenemos que implementar la función si esta lleva el nombre `findBy<atributo>`, se implementa de forma automática en el `CrudRepository`).

```
public interface UsersRepository extends CrudRepository<User, Long>{  
  
    User findByDni(String dni);  
  
}
```

Implementamos añadiendo un método en **UserService** que nos permita obtener un usuario por Dni (si no está definido anteriormente), llamará a la función que acabamos de implementar en el repositorio.

```
public User getUserByDni(String dni) {  
    return usersRepository.findByDni(dni);  
}
```



Revisamos en **signup.html** todos los nombres de todos los campos del formulario, son los siguientes: **dni, name, lastName, password, passwordConfirm**

Incluimos la validación de todos estos campos en **SignUpFormValidator**.

La función **validate()** recibe el **target** (objeto con los datos del formulario), se comprueba cada atributo, si existe algún error incluimos información sobre él en el objeto **errors**. Para cada error incluimos el nombre del campo relacionado con el error y la ID del mensaje (fichero messages de internacionalización) que queremos mostrar como error.

La clase **ValidationUtils** implementa algunas comprobaciones comunes como comprobar si un campo está vacío, si queremos hacer comprobaciones más específicas debemos implementarlas.

```
import com.uniovi.entities.User;  
import com.uniovi.services.UserService;  
import org.springframework.beans.factory.annotation.Autowired;  
import org.springframework.stereotype.Component;  
import org.springframework.validation.*;  
  
@Component  
public class SignUpFormValidator implements Validator {  
  
    @Autowired  
    private UserService userService;  
  
    @Override  
    public boolean supports(Class<?> aClass) {
```



```
        return User.class.equals(aClass);
    }

    @Override
    public void validate(Object target, Errors errors) {
        User user = (User) target;
        ValidationUtils.rejectIfEmptyOrWhitespace(errors, "dni", "Error.empty");

        if (user.getDni().length() < 5 || user.getDni().length() > 24) {
            errors.rejectValue("dni", "Error.signup.dni.length");
        }

        if (userService.getUserByDni(user.getDni()) != null) {
            errors.rejectValue("dni", "Error.signup.dni.duplicate");
        }

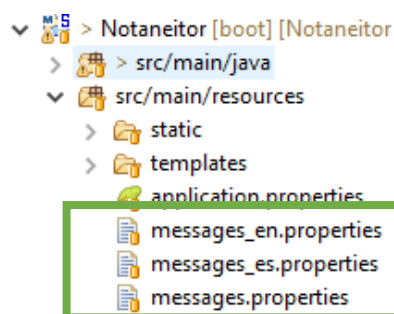
        if (user.getName().length() < 5 || user.getName().length() > 24) {
            errors.rejectValue("name", "Error.signup.name.length");
        }

        if (user.getLastName().length() < 5 || user.getLastName().length() > 24) {
            errors.rejectValue("lastName", "Error.signup.lastName.length");
        }

        if (user.getPassword().length() < 5 || user.getPassword().length() > 24) {
            errors.rejectValue("password", "Error.signup.password.length");
        }

        if (!user.getPasswordConfirm().equals(user.getPassword())) {
            errors.rejectValue("passwordConfirm",
                "Error.signup.passwordConfirm.coincidence");
        }
    }
}
```

Declaramos todos los mensajes en los ficheros **messages\_en.properties** para inglés y **messages.properties** y **messages\_es.properties** para Español.



### messages\_en.properties

```
Error.empty= This field is required.
Error.signup.dni.length=The DNI must have between 5 and 24 characters.
Error.signup.dni.duplicate=This DNI already exist.
Error.signup.name.length=The name must have between 5 and 24 characters.
Error.signup.lastName.length=The last name must have between 5 and 24 characters.
Error.signup.password.length = The password must have between 5 and 24 characters.
Error.signup.passwordConfirm.coincidence=These passwords don't match.
```

### messages.properties y messages\_es.properties



```
Error.empty= Este campo no puede ser vacío  
Error.signup.dni.length=El DNI debe tener entre 5 y 24 caracteres.  
Error.signup.dni.duplicate=This DNI already exist.  
Error.signup.name.length=El nombre debe tener entre 5 y 24 caracteres.  
Error.signup.lastName.length=El apellido debe tener entre 5 y 24 caracteres.  
Error.signup.password.length =La contraseña debe tener entre 5 y 24 caracteres.  
Error.signup.passwordConfirm.coincidence=Las contraseñas no coinciden
```

Modificamos la vista **signup.html** para configurar el sistema de validación.

- En la declaración del form incluimos el **th:object** (objeto a validar) en este caso **#{user}** un usuario
- Junto a cada input incluimos una etiqueta **span** de clase **text-danger**, esta etiqueta solo se va a mostrar si hay un error en el input, podemos evaluarlo con **th:if**, el texto de la etiqueta de spam viene dado por **th:errors**. Cada etiqueta **span** nuestra comprueba los errores de su campo específico, **el dni, name, username, etc.**

```
<h2>Regístrate como usuario</h2>  
<form class="form-horizontal" method="post" action="/signup" th:object="#{user}">  
  <div class="form-group">  
    <label class="control-label col-sm-2" for="dni">DNI:</label>  
    <div class="col-sm-10">  
      <input type="text" class="form-control" name="dni"  
        placeholder="99999999Y" required="true" />  
      <span class="text-danger" th:if="#{fields.hasErrors('dni')}"  
        th:errors="#{dni}">  
    </div>  
    </div>  
  <div class="form-group">  
    <label class="control-label col-sm-2" for="name">Nombre:</label>  
    <div class="col-sm-10">  
      <input type="text" class="form-control" name="name"  
        placeholder="Ejemplo: Juan" required="true" />  
      <span class="text-danger" th:if="#{fields.hasErrors('name')}"  
        th:errors="#{name}" />  
    </div>  
    </div>  
  <div class="form-group">  
    <label class="control-label col-sm-2" for="lastName">Apellidos:</label>  
    <div class="col-sm-10">  
      <input type="text" class="form-control" name="lastName"  
        placeholder="Ejemplo: Pérez Almonte" required="true" />  
      <span class="text-danger" th:if="#{fields.hasErrors('lastName')}"  
        th:errors="#{lastName}" />  
    </div>  
    </div>  
  <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="Entre el Password" />  
      <span class="text-danger" th:if="#{fields.hasErrors('password')}"  
        th:errors="#{password}" />  
    </div>  
    </div>  
  <div class="form-group">  
    <label class="control-label col-sm-2" for="passwordConfirm">Repita  
    el Password:</label>  
    <div class="col-sm-10">  
      <input type="password" class="form-control" name="passwordConfirm"  
        placeholder="Repita el Password" />  
      <span class="text-danger" th:if="#{fields.hasErrors('passwordConfirm')}"  
        th:errors="#{passwordConfirm}" />  
    </div>  
    </div>  
</form>
```



```
</div>
</div>
<div class="form-group">
  <div class="col-sm-offset-2 col-sm-10">
    <button type="submit" class="btn btn-primary">Enviar</button>
  </div>
</div>
</form>

</div>
```

Finalmente debemos modificar el **UsersController**, la respuesta **GET/signup** que retornada la vista **signup** espera recibir un parámetro **user**, con un usuario vacío (sin datos inicialmente). Este objeto se está solicitando el **th:object="\${user}"** que hemos incluido en la vista.

```
@RequestMapping(value = "/signup", method = RequestMethod.GET)
public String signup(Model model) {
    model.addAttribute("user", new User());
    return "signup";
}
```

La parte que realmente realiza la validación será **POST/signup**, es la que recibe los datos del formulario y debe validarlos. En primer lugar inyectamos el Componente **SignUpFormValidator**

```
@Controller
public class UsersController {

    @Autowired
    private UsersService usersService;

    @Autowired
    private SecurityService securityService;

    @Autowired
    private SignUpFormValidator signUpFormValidator;
```


En **POST/signup** modificamos los parámetros de entrada. Aplicamos el **signUpFormValidator**, si se produce un error redirigimos a la vista **signup**, es la vista actual (no se ha podido completar el registro)

```
@RequestMapping(value = "/signup", method = RequestMethod.POST)
public String signup(@Validated User user, BindingResult result, Model model) {
    signUpFormValidator.validate(user, result);
    if (result.hasErrors()) {
        return "signup";
    }

    usersService.addUser(user);
    securityService.autoLogin(user.getDni(), user.getPasswordConfirm());
    return "redirect:home";
}
```



}

 Home Gestión de notas ▼ Gestión de Usuarios ▼ Idioma ▼ Registrarse Identificarse

### Regístrate como usuario

DNI:

El DNI debe tener entre 5 y 24 caracteres.

Nombre:

El nombre debe tener entre 5 y 24 caracteres.

Apellidos:

El apellido debe tener entre 5 y 24 caracteres.

Password:

La contraseña debe tener entre 5 y 24 caracteres.

Repita el Password:

Enviar

## 5.1 Ejercicio propuesto

Realizar validación en el servidor de los formularios disponibles en la web.