Sistemas Distribuidos e Internet

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

Sesión 4

Curso 2023/2024

Contenido

¡¡¡¡¡¡MUY IMPORTANTE!!!!!!

Como esta práctica guiada es una continuación de la anterior, seguiremos utizando el mismo repositorio de Github de la práctica anterior.

1 Thymeleaf – fragmentos

Thymeleaf ofrece la posibilidad de actualizar únicamente 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="marksTable"**, también deberíamos darle una id para poder acceder a él de forma rápida desde jQuery. Para realizar una actualización parcial de la lista de notas realizamos los siguientes pasos:

Definir endpoint en el controlador

Lo primero que haremos es definir un endpoint (URL) que devuelva el fragmento correspondiente, en lugar de la vista completa. En el controlador **MarkControllers** definimos la url **/mark/list/update**. La única diferencia respecto a mark/list es que no retorna toda la vista, solamente el fragmento **marksTable.**

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

Actualizamos la vista

Modificamos la siguiente línea en el **fichero list.html** para que solo se actualice el fragmento correspondiente de la vista.

```
<div class="container" id="main-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="marksTable" id="marksTable">
         <thead>
         <tr>
            <th scope="col">id</th>
```

Añadir un botón que llame el método actualizar del controlador.

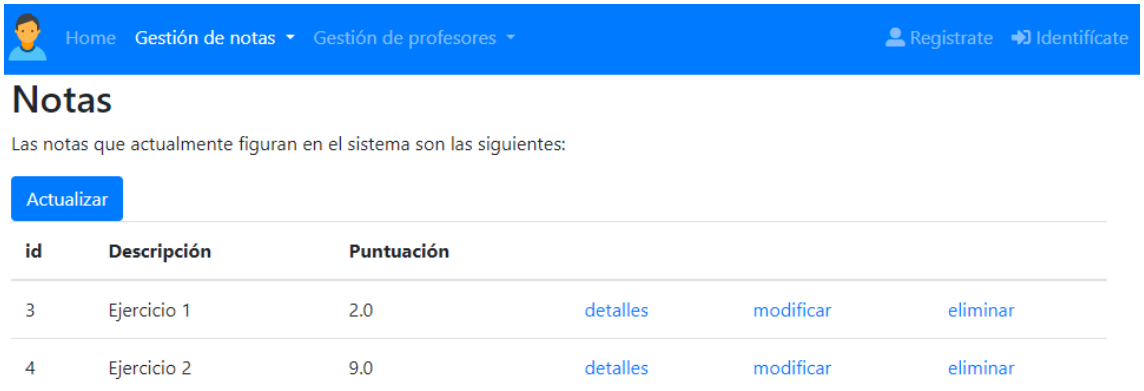
Finalmente, solo tenemos que incluir un botón que realice la llamada a **/list/update** y sustituya el contenido de la tabla con id **marksTable**. (En lugar de un botón también podríamos seguir otras estrategias: cada N segundos, o cada vez que la tabla reciba el foco, etc.)

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

```
<div class="container" id="main-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-primary">Actualizar</button>
  <script>
    $( "#updateButton" ).click(function() {
      $("#marksTable").load('/mark/list/update');
    });
  </script>
  <div class="table-responsive">
    <table class="table table-hover" th:fragment="marksTable" id="marksTable">
```

Probar la funcionalidad añadida

El nuevo botón "Actualizar", recargará únicamente la tabla (probamos a crear una **nueva nota desde otra pestaña y a actualizar la lista**). Desplegamos la aplicación y probamos.



Nota: Incluir el siguiente Commit Message ->

"SDI-IDGIT-3.1-Thymeleaf, fragmentos."

OJO: sustituir IDGIT por tu número asignado (p.e. 2324-101):

"SDI-2324-101-3.1-Thymeleaf, fragmentos."

(*No olvides incluir los guiones y NO incluyas BLANCOS al pricinpio o al final de la oración*)

2 Internacionalización

En este apartado veremos cómo podemos añadir internacionalización usando Thymeleaf.

Para configurar nuestra aplicación web para que soporte internacionalización, se deben realizar las siguientes modificaciones en la aplicación.

2.1 Modificar la configuración de Spring Boot

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

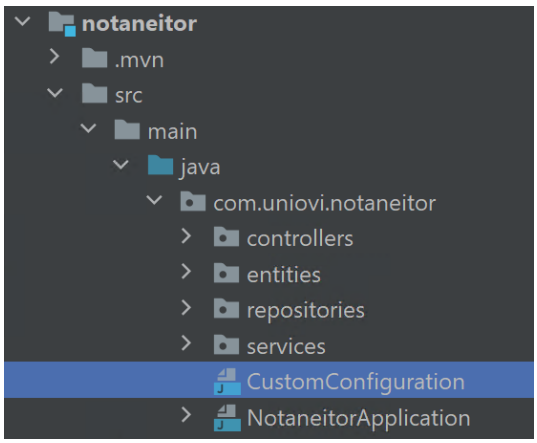
LocaleResolver → **detectar localización.** Para que nuestra aplicación pueda determinar la localización que se está utilizando actualmente, necesitamos agregar un Bean ***LocaleResolver***.

La interfaz de ***LocaleResolver*** tiene implementaciones que determinan el entorno local actual basado en la sesión, las cookies, la cabecera 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 el 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 de nuestra aplicación. Hasta ahora, estabámos 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*** que implementará la interfaz ***WebMvcConfigurer*** en nuestro proyecto:



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

```
package com.uniovi.notaneitor;

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.WebMvcConfigurer;
import org.springframework.web.servlet.i18n.LocaleChangeInterceptor;
import org.springframework.web.servlet.i18n.SessionLocaleResolver;


@Configuration
public class CustomConfiguration implements WebMvcConfigurer {

   @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

Lo siguiente es definir las fuentes de los mensajes (palabras, frases, oraciones…) en los distintos idiomas que soportará la aplicación y que se mostrarán a los usuarios en función del idioma elegido.

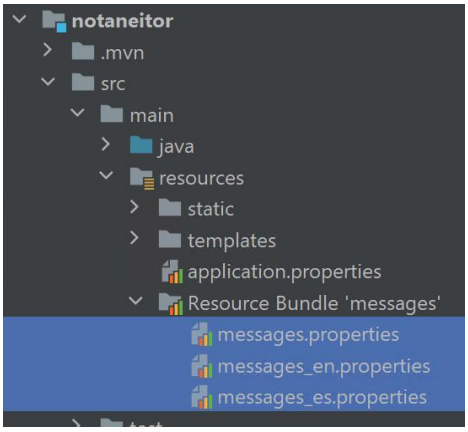
De forma predeterminada, la aplicación buscará archivos de mensajes que contengan las claves y valores de internacionalización en la carpeta ***src/main/resources.***

El archivo de la configuración del idioma local predeterminada tendrá el nombre ***messages.properties*** y los archivos de cada idioma que soporte la 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 devolverá 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=Inglés
language.es=Español
login.message=Identifícate
signup.message=Regístrate
```

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
```

2.3 Modificar las vistas

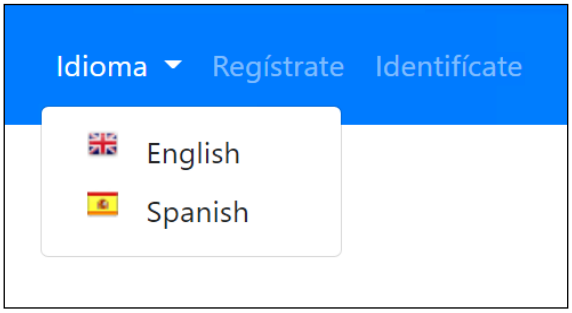
En el motor ***Thymeleaf*** accedemos a los valores de las variables usando las claves, con la sintaxis ***#{clave}.***

A continuación, vamos a modificar la página ***index.html*** para internacionalizar el mensaje de bienvenida.

```
<body>
  <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"/>
</body>
```

Añadir un menú de idioma en la barra de navegación

Ahora vamos a modificar la barra de navegación de la aplicación, añadiendo el siguiente menú desplegable, para que el usuario pueda cambiar el idioma por defecto si lo desea:

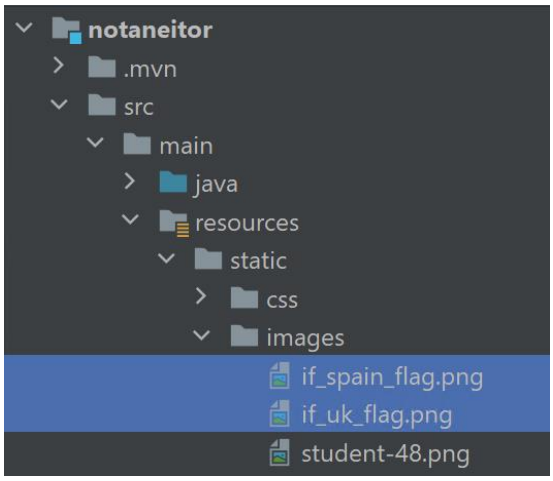


A continuación, realizamos los siguientes cambios en la página ***fragments/nav.html***:

Primero, 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 ***images*** del proyecto. **Estas imágenes están disponibles en los recursos (zip) de la práctica.**

Es importante que no arrastréis o copiéis y peguéis las imagénes en la carpeta corresponidente usando el IDE. Es probable que, a la hora de compilar, IntelliJ no detecte correctamente esos ficheros.

Copiad los ficheros mediante el explorador de archivos en la carpeta correspondiente. Seguidamente, **clic derecho en la carpeta images del IDE → Reload from disk.**

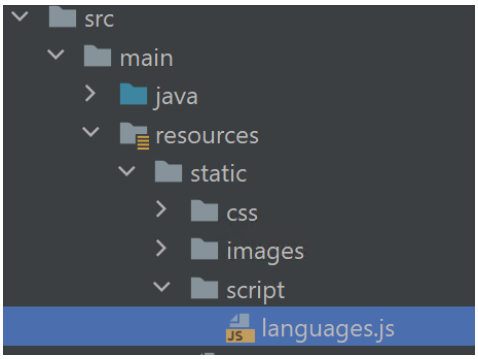


Seguidamente, en el mismo fichero **fragments/*nav.html*,** modificamos los textos de algunas de las opciones de menú, para que cambien en función del idioma.

```
<!-- nav.html -->
<nav class="navbar navbar-expand-lg navbar-dark bg-primary">
  <a class="navbar-brand" href="#"></a>
  <img src="/images/student-48.png" alt="logo"/>
  <button class="navbar-toggler" type="button" data-toggle="collapse" data-target="#mynavbar"
        aria-controls="navbarColor02" aria-expanded="false" aria-label="Toggle navigation">
    <span class="navbar-toggler-icon"></span>
  </button>
  <div class="collapse navbar-collapse" id="mynavbar">
    <ul class="navbar-nav mr-auto">
      <li class="nav-item ">
        <a class="nav-link" href="/home">Home<span class="sr-only">(current)</span></a>
      </li>
      <li class="nav-item dropdown active">
        <a class="nav-link dropdown-toggle" href="#" id="navbarDropdown" role="button" data-toggle="dropdown"
          aria-haspopup="true" aria-expanded="false"> Gestión de notas
        </a>
        <div class="dropdown-menu" aria-labelledby="navbarDropdown">
          <a class="dropdown-item" href="/mark/list">Ver Notas</a>
          <a class="dropdown-item" href="/mark/add">Agregar Nota</a>
          <div class="dropdown-divider"></div>
          <a class="dropdown-item" href="/mark/filter">Filtrar</a>
        </div>
      </li>
      <li class="nav-item dropdown">
        <a class="nav-link dropdown-toggle" href="#" id="profesorDropdown" role="button" data-toggle="dropdown"
          aria-haspopup="true" aria-expanded="false"> Gestión de profesores
        </a>
        <div class="dropdown-menu" aria-labelledby="navbarDropdown">
          <a class="dropdown-item" href="/professor/list">Ver profesores</a>
        </div>
      </li>
    </ul>
    <ul class="navbar-nav justify-content-end">
      <li class="nav-item dropdown">
        <a class="nav-link dropdown-toggle" href="#" id="btnLanguage" role="button" data-toggle="dropdown"
          aria-haspopup="true" aria-expanded="false">
          <span th:text="#{language.change}"></span>
        </a>
```

```
          <div id="languageDropdownMenuButton" class="dropdown-menu" aria-labelledby="navbarDropdown">
            <a class="dropdown-item" id="btnEnglish" value="EN">
              <img th:src="@{/images/if_uk_flag.png}" style="border: 0; float: left; margin-right: 15px"/>
              <span th:text="#{language.en}">Inglés</span>
            </a>
            <a class="dropdown-item" id="btnSpanish" value="ES">
              <img th:src="@{/images/if_spain_flag.png}" style="border: 0; float: left; margin-right: 15px"/>
              <span th:text="#{language.es}">Español</span>
            </a>
          </div>
        </li>
        <li class="nav-item">
          <a class="nav-link" href="/signup">
            <i class='fas fa-user-alt' style='font-size:16px'></i>
            <span th:text="#{signup.message}"></span>
          </a>
        </li>
        <li class="nav-item">
          <a class="nav-link" href="/login">
            <i class='fas fa-sign-in-alt' style='font-size:16px'></i>
            <span th:text="#{login.message}"></span>
          </a>
        </li>
      </ul>
    </div>
</nav>
```

Para finalizar, el selector de idioma no es funcional, por lo que tenemos que incluir un script quie permita realizar el cambio de idioma. Para ello, 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***.



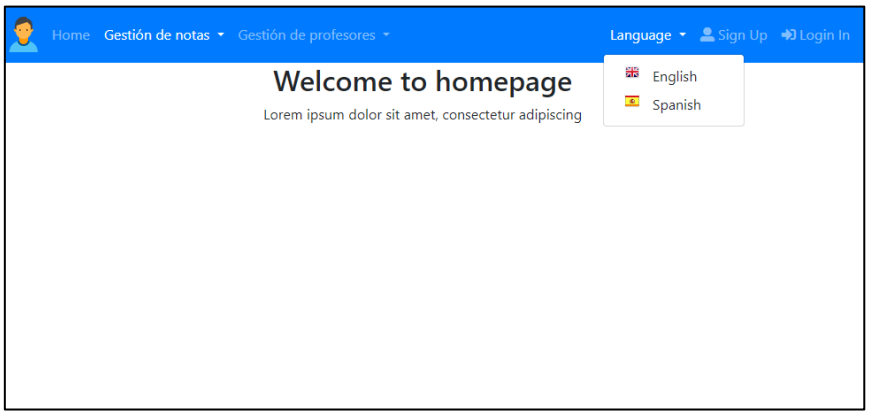

```
$(document).ready(function () {
    $("#languageDropdownMenuButton a").click(function (e) {
        e.preventDefault(); // cancel the link behaviour
        let languageSelectedText = $(this).text();
        let languageSelectedValue = $(this).attr("value");

        $("#btnLanguage").text(languageSelectedText);
        window.location.replace('?lang=' + languageSelectedValue);
        return false;
    });
});
```

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

```
<!-- head.html -->
<head>
  <title>Notaneitor</title>
  <meta charset="utf-8"/>
  <meta name="viewport" content="width=device-width, initial-scale=1"/>
  <!-- Bootstrap CSS -->
  <link rel="stylesheet" href="https://maxcdn.bootstrapcdn.com/bootstrap/4.5.2/css/bootstrap.min.css">
  <!-- Font Awesome CSS -->
  <link rel="stylesheet" href="https://use.fontawesome.com/releases/v5.7.0/css/all.css"
      integrity="sha384-lZN37f5QGtY3VHgisS14W3ExzMWZxybE1SJSEsQp9S+oqd12jhcu+A56Ebc1zFSJ" crossorigin="anonymous">
  <!-- JS files: jQuery primero y luego Bootstrap JS -->
  <script src="https://ajax.googleapis.com/ajax/libs/jquery/3.5.1/jquery.min.js"></script>
  <script src="https://maxcdn.bootstrapcdn.com/bootstrap/4.5.2/js/bootstrap.min.js"></script>
  <!-- Nuestra CSS -->
  <link rel="stylesheet" href="/css/custom.css" />
  <script src="/script/languages.js"> </script>
</head>
```

Probar la nueva funcionalidad

Deplegamos la aplicación y aparecerá un menú deplegable de idioma con dos posibles opciones. Cuando se ha 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.



Nota: Incluir el siguiente Commit Message ->

"SDI-IDGIT-3.2-Internacionalización."

OJO: sustituir IDGIT por tu número asignado (p.e. 2324-101):

"SDI-2324-101-3.2-Internacionalización."

(*No olvides incluir los guiones y NO incluyas BLANCOS al pricinpio o al final de la oración*)

3 Acceso a datos con entidades relacionadas

Vamos a incluir **usuarios** con dos roles (roles: alumno y profesor). Tambien se establecerá una relaccion 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.

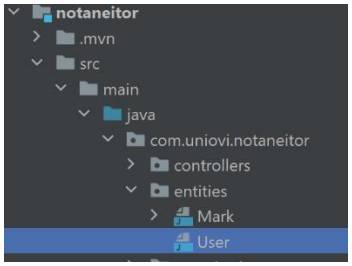
3.1 Modelos de datos

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*** para indicar 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***[1] se puede utilizar en relacciones ***OneToMany o ManyToMany para*** indicar que la estructura de datos se carga de forma proactiva (en lugar de perezosa) en el mismo momento que se carga la entidad User.




```java
package com.uniovi.notaneitor.entities;

import javax.persistence.*;
import java.util.Set; //Colección que no admite duplicados

@Entity
public class User {

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

[1] https://www.baeldung.com/hibernate-lazy-eager-loading

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

    public void setMarks(Set<Mark> marks) {
        this.marks = 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 muchos a uno con la entidad usuarios (User) usando la anotación ***@ManyToOne*** y mediante la columna ***user_id*** mediante la anotación ***@JoinColumn*** que especifica la columna que va a crear una asociación entres las entidades.
- Creamos un nuevo constructor donde reciba un usuario.
- Creamos métodos get y set para usuario.

```
package com.uniovi.notaneitor.entities;

import javax.persistence.*;

@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() {
  }

  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;
  }

  @Override
  public String toString() {
    return "Mark{" + "id=" + id + ", description='" + description + '\'' + ", score=" + score + '}';
  }

  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 qué 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 especifica. Si se omite @ Column, se utiliza el valor predeterminado: el nombre de campo de la entidad.

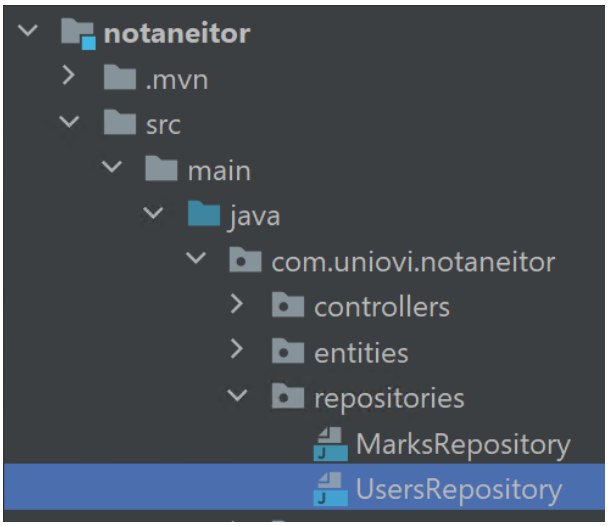
@OneToMany: Define una asociación de muchos valores con multiplicidad de uno a muchos.

@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

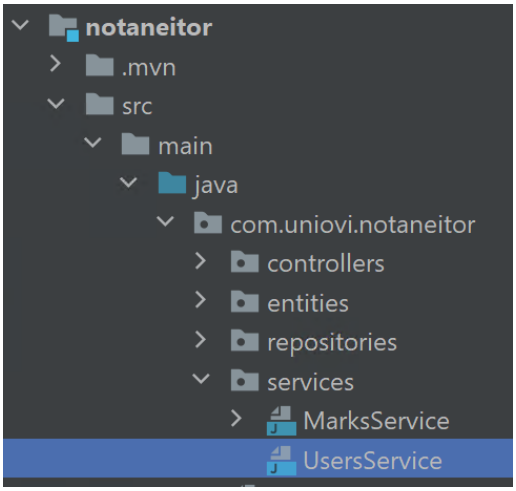
Al igual que hicimos previamente para la entidad **Mark**, ahora debemos crear un repositorio ***JPA*** para la entidad ***User***, la interface se llamará **UsersRepository** y va a heredar de CrudRepository.




```
package com.uniovi.notaneitor.repositories;

import com.uniovi.notaneitor.entities.*;
import org.springframework.data.repository.CrudRepository;

public interface UsersRepository extends CrudRepository<User, Long>{
}
```

3.3 Definir servicios

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




```
package com.uniovi.notaneitor.services;

import java.util.*;
import javax.annotation.PostConstruct;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Service;
import com.uniovi.notaneitor.entities.*;
import com.uniovi.notaneitor.services.UsersService;

@Service
public class UsersService {
  private final UsersRepository usersRepository;

public UsersService(UsersRepository usersRepository) {
   this.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.findById(id).get();
   }

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

   public void deleteUser(Long id) {
```

```
    usersRepository.deleteById(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.notaneitor.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.notaneitor.entities.Mark;
import com.uniovi.notaneitor.entities.User;

@Service
public class InsertSampleDataService {

private final UsersService usersService;

public InsertSampleDataService(UsersService usersService) {
  this.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 crecación de la base de datos.

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

Si quisiéramos desactivar el servicio bastaría con eliminar la anotación etiqueta **@Service**

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

3.4 Definir y actualizar controladores

Primero vamos a actualizar 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 a 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 a las vistas. Por tanto, la vista permitirá seleccionar un usuario de la aplicación.

- En **POST /mark/edit/<id>** solamente vamos a permitir modificar los atributos **score y description**, no se podrá por lo tanto modificar el atributo usuario.

Obtenemos nota original correspondiente a la **<id>,** sobreescribimos los atributos **score** y **description**; finalmente salvamos el objeto en el repositorio.

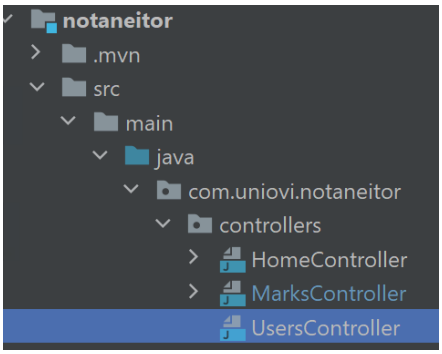
Para hacer estos cambios vamos a realizar dos pasos:

- Inyectar el servicio UserService en el controlador
- Modificar los métodos: mark/add, mark/edit (GET y POST)

Nota: Modificar sólo los métodos que cambian. Todos los demás métodos no cambian.

```
// Inyectamos el servicio por inyección basada en constructor
private final MarksService marksService;
 private final UsersService usersService;
public MarksController(MarksService marksService, UsersService usersService) {
   this.marksService = marksService;
   this.usersService = usersService;
}

// Modificamos los siguientes metodos
@RequestMapping(value="/mark/add")
public String getMark(Model model){
   model.addAttribute("usersList", usersService.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", usersService.getUsers());
   return "mark/edit";
}

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

Ahora vamos a definir un nuevo controlador para realizar la gestión de usuarios. Creamos una nueva clase **UsersController**:



Este controlador será prácticamente igual qie el controlador de las **notas**, pero basado en los **usuarios, y permitirá**: listar, añadir, ver detalles, modificar y eliminar.

```
package com.uniovi.notaneitor.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.notaneitor.entities.*;
import com.uniovi.notaneitor.services.UsersService;

@Controller
public class UsersController {

  private final UsersService usersService;

  public UsersController(UsersService usersService) {
    this.usersService = usersService;
  }

  @RequestMapping("/user/list")
  public String getListado(Model model) {
    model.addAttribute("usersList", usersService.getUsers());
    return "user/list";
  }

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

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

```
  @RequestMapping("/user/details/{id}")
  public String getDetail(Model model, @PathVariable Long id) {
    model.addAttribute("user", usersService.getUser(id));
    return "user/details";
  }
  @RequestMapping("/user/delete/{id}")
  public String delete(@PathVariable Long id) {
    usersService.deleteUser(id);
    return "redirect:/user/list";
  }

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

  @RequestMapping(value = "/user/edit/{id}", method = RequestMethod.POST)
  public String setEdit(@PathVariable Long id, @ModelAttribute User user) {
    usersService.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**, realizando las siguientes modificaciones.

Añadir un menú de gestión de usuarios

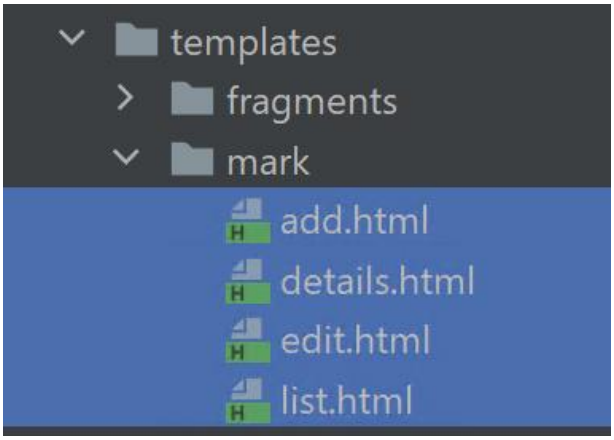
Comenzamos modificando el fragmento de la barra de navegación **fragments/nav.html.**

Par llevar a cabo esta modificación debemos ***eliminar la opción de filtrar y añadir un dropdown*** (menú desplegable) para la gestión de **usuarios** en el fichero nav.html

```
<ul class="navbar-nav mr-auto">
  <li class="nav-item ">
    <a class="nav-link" href="/home">Home<span class="sr-only">(current)</span></a>
  </li>
  <li class="nav-item dropdown active">
    <a class="nav-link dropdown-toggle" href="#" id="navbarDropdown" role="button" data-
toggle="dropdown"
      aria-haspopup="true" aria-expanded="false"> Gestión de notas
    </a>
    <div class="dropdown-menu" aria-labelledby="navbarDropdown">
      <a class="dropdown-item" href="/mark/add">Agregar Nota</a>
      <a class="dropdown-item" href="/mark/list">Ver Notas</a>
      <div class="dropdown-divider"></div>
      <a class="dropdown-item" href="/mark/filter">Filtrar</a>
    </div>
  </li>

  <li class="nav-item dropdown">
    <a class="nav-link dropdown-toggle" href="#" id="userDropdown" role="button" data-
toggle="dropdown"
      aria-haspopup="true" aria-expanded="false"> Gestión de usuarios
    </a>
    <div class="dropdown-menu" aria-labelledby="navbarDropdown">
      <a class="dropdown-item" href="/user/add">Agregar usuario</a>
      <a class="dropdown-item" href="/user/list">Ver usuarios</a>
    </div>
  </li>
```

Modificar las vistas de gestión de notas

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.

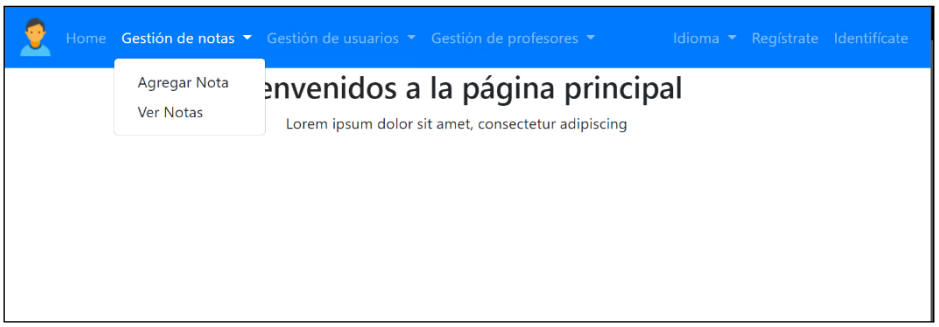
```
<form class="form-horizontal" method="post" action="/mark/add">
 <div class="form-group">
  <label class="control-label col-sm-2" for="user">Alumno:</label>
  <div class="col-sm-10" th:if="${usersList != null and not #lists.isEmpty(usersList)}">
   <select id="user" class="form-control" name="user">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

En la página **details.html**, añadimos el atributo **mark.user.dni**

```
<div class="container" id="main-container">
  <h2>Detalles de la nota</h2>
  <div class="card">
    <div class="card-header">DNI</div>
    <div class="card-body">
      <p class="card-text" th:text="${mark.user.dni}">88888847X</p>
    </div>
  </div>
  <br>
```

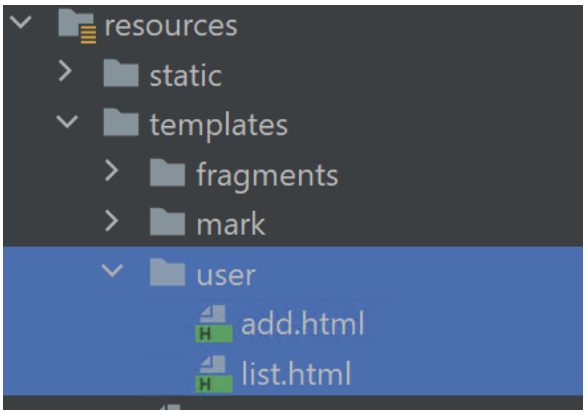
Probar la nueva funcionalidad

Llegados a este punto, ya podemos probar las funcionalidades dependientes de **Nota**.



Crear las vistas para de gestión de usuarios

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



La página **user/add.html** nos permitirá registrar un usuario mediante un formulario.

```
<!DOCTYPE html>
<html lang="en" xmlns="http://www.w3.org/1999/xhtml" xmlns:th="http://thymeleaf.org">
<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" id="dni" 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" id="name" 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" id="lastName" name="lastName"
placeholder="Ejemplo: Peréz 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>
```

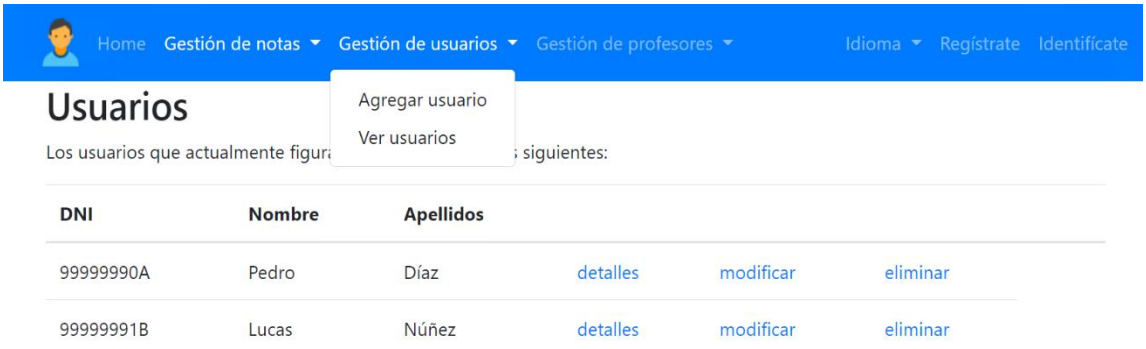
```
    </div>
  </form>
</div>
<footer th:replace="fragments/footer"/>
</body>
</html>
```

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

```
<!DOCTYPE html>
<html lang="en" xmlns="http://www.w3.org/1999/xhtml" xmlns:th="http://thymeleaf.org">
<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 scope="col">DNI</th>
        <th scope="col">Nombre</th>
        <th scope="col">Apellidos</th>
        <th scope="col"></th>
        <th scope="col"></th>
        <th scope="col"></th>
        <th scope="col"></th>
        <th scope="col"></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>
```

Probar la nueva funcionalidad

Desplegamos de nuevo la aplicación y comprobamos que la aplicación nos permite **listar**, **crear y eliminar** usuarios correctamente.



Notas:

- 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" en cada ejecución.
- Si en algún momento aparecen problemas para crear la base de datos, borrar su schema manualmente (podemos hacerlo mediante comandos o volviendo a extraer el contenido de **hsqldb.zip**).

Nota: Incluir el siguiente Commit Message ->

"SDI-IDGIT-3.3-Acceso a datos con entidades relacionadas."

OJO: sustituir IDGIT por tu número asignado (p.e. 2324-101):

"SDI-2324-101-3.3-Acceso a datos con entidades relacionadas."

(*No olvides incluir los guiones y NO incluyas BLANCOS al pricinpio o al final de la oración*)

4 Autenticación y control de acceso

La autenticación y control de acceso a recursos en las aplicaciones web puede ser un punto crítico en muchas aplicaciones, ya que la falta de estos servicios podría poner en riesgo la seguridad de una aplicación. 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,** un framework (spring security) que se centra en proporcionar autenticación y autorización para aplicaciones web de una manera comprensible y extensible.

Segundo, añadir la dependencia ***thymeleaf-extras-springsecurity4,*** un módulo extra que no es parte del core de Thymeleaf. Proporciona un conjunto de elementos para trabajar con los objetos de spring security de una manera sencilla desde las plantillas Thymeleaf[2].

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

<dependency>
  <groupId>org.thymeleaf.extras</groupId>
  <artifactId>thymeleaf-extras-springsecurity4</artifactId>
  <version>3.0.4.RELEASE</version>
</dependency>
```

Una vez añadidas estas dependencias hay que actualizar las dependencias desde el panel de Maven en IntelliJ. De esta forma Spring Security ya estará siendo utilizada en la aplicación.

4.2 Definición del atributo password

Para autenticarnos en la aplicación web necesitaremos unas credenciales (usuario y contraseña). Vamos a modificar la aplicación para que se pueda llevar a cabo esta acción.

Lo primero que haremos es modificar la entidad **User**, añadiendo 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
```

[2] A veces, Maven no puede descargar la última versión de **spring-boot-starter-security**, por ello se recomienda incluir una de las versions anteriores (p.e. **<version>2.1.0.RELEASE</version>)** actualizar maven y volver a suprimir la versión para que Maven actualice a la última versión.

```
  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;
```

A continuación, añadimos los **getters** y **setters** correspondientes para poder obtener y establecer los valore de estos atibutos:

```
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

Para poder persistir los nuevos datos tenemos que modificar el **UsersRepository.** En esta aplicación vamos a utilizar el **DNI** para identificar a los usuarios, en lugar de un nombre de usuario (username), por lo que, tenemos que definir un método ***findByDni()*** para buscar usuario por DNI. ***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

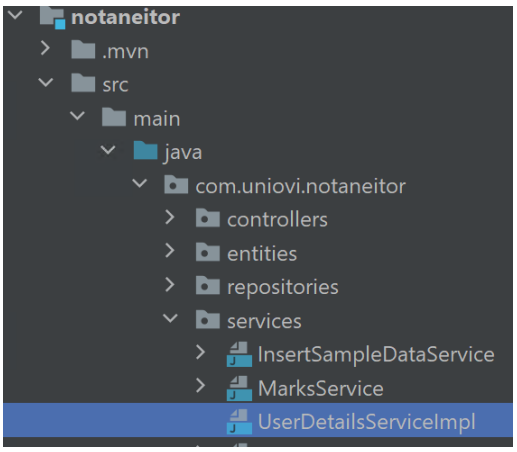
A continuación, vamos a definir los distintos servicios que tendrán toda la lógica de negocio del sistema de autenticación y control de acceso de la aplicación.

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 este método debemos devolver un user **(security.core.userdetails.User)** y sus roles, en nuestro caso el Username es el dni del usuario (aunque podríamos haber utilizado otro valor único: la id, el UO, etc.)



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

Nota 2: Spring security llama a las propiedades de identificación: Username y password.

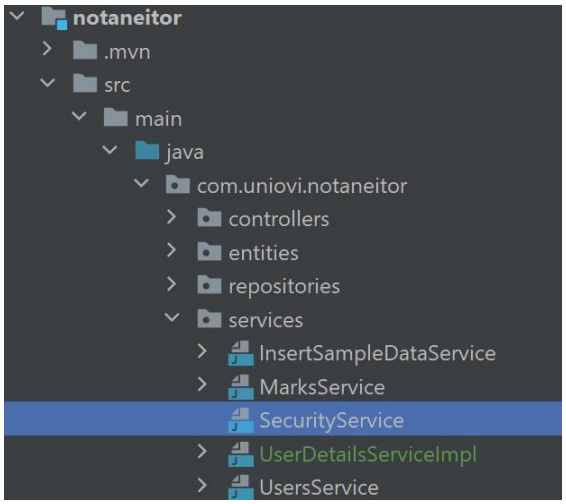
Obtenemos el usuario de nuestra aplicación y creamos un userdetails con su DNI, password y roles del usuario.

```
package com.uniovi.notaneitor.services;

import org.springframework.stereotype.Service;
import com.uniovi.notaneitor.entities.User;
import com.uniovi.notaneitor.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 java.util.*;

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

   public UserDetailsServiceImpl(UsersRepository usersRepository) {
      this.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_STUDENT"));
      if (user == null) {
           throw new UsernameNotFoundException(dni);
      }
      return new org.springframework.security.core.userdetails.User(
            user.getDni(), user.getPassword(), grantedAuthorities);
   }
}
```

4.4.2 Servicio SecurityService

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




```java
package com.uniovi.notaneitor.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 {
   private static final Logger logger = LoggerFactory.getLogger(SecurityService.class);
   private final AuthenticationManager authenticationManager;
   private final UserDetailsService userDetailsService;

    public SecurityService(AuthenticationManager authenticationManager, UserDetailsService
userDetailsService) {
     this.authenticationManager = authenticationManager;
     this.userDetailsService = userDetailsService;
 }

   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));
    }
  }
}
```

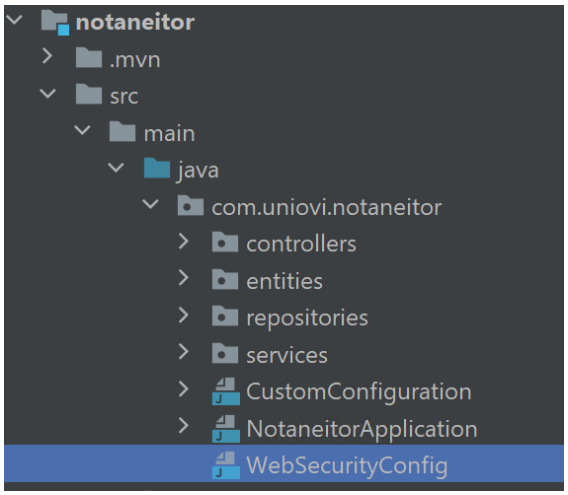
Nota: En este punto es probable que nos dé un error de que no puede inyectar el bean AutenticationManager. **"Could not autowire. No beans of 'AuthenticationManager' type found."**. Para correir este error, debemos definir un adaptador de seguridad que luego utilizaremos más adelante para configurar algunos mecanismos de seguridad.

4.4.3 Configuración del adaptador de seguridad

Crearemos una nueva configuración de seguridad denominada **WebSecurityConfig** en el paquete ***com.uniovi.notaneitor.*** La clase **WebSecurityConfig** heredará de ***WebSecurityConfigurerAdapter*** e incluirá la anotación ***@EnableWebSecurity***.

La anotación @EnableWebSecurity y WebSecurityConfigurerAdapter funcionan juntos para proporcionar seguridad basada en web. Al extender WebSecurityConfigurerAdapter y añadiendo solo unas pocas líneas de código, podemos hacer lo siguiente:

- Requerir que el usuario esté autenticado antes de acceder a cualquier URL dentro de nuestra aplicación.
- Crear un usuario con el nombre de usuario "user", la contraseña "password" y el rol de "ROLE_USER".
- Habilitar la autenticación HTTP básica y la basada en formularios.
- Spring Security generará automáticamente una página de inicio de sesión (login) y una página de cierre de sesión (logout).



Vamos a incluir dos beans en esta clase:

- **AuthenticationManager**: lo utilizaremos para definir cómo los filtros de Spring Security llevarán a cabo la autenticación en nuestra aplicación web.
- **BCryptPasswordEncoder**: lo utilizaremos para cifrar las contraseñas en la aplicación.

```
package com.uniovi.notaneitor;

import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import org.springframework.security.authentication.AuthenticationManager;
import org.springframework.security.config.annotation.web.configuration.EnableWebSecurity;
import org.springframework.security.config.annotation.web.configuration.WebSecurityConfigurerAdapter;
import org.springframework.security.crypto.bcrypt.BCryptPasswordEncoder;

@Configuration
@EnableWebSecurity
public class WebSecurityConfig extends WebSecurityConfigurerAdapter {

   @Bean
   @Override
   public AuthenticationManager authenticationManagerBean() throws Exception {
      return super.authenticationManagerBean();
   }

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

Nota: En este punto se habrá eliminado el error de inyección comentado anteriormente.

4.4.4 Actualizar el servicio UserService

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

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

Modificamos el método **addUser()** para que al guardar un objeto usuario ***cifre el password,*** utilizando el bean **BCryptPasswordEncoder** definido en el paso anterior.

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

```
import org.springframework.security.crypto.bcrypt.BCryptPasswordEncoder;

@Service
public class UsersService {
   private final UsersRepository usersRepository;

   private final BCryptPasswordEncoder bCryptPasswordEncoder;

   public UsersService(UsersRepository usersRepository, BCryptPasswordEncoder
bCryptPasswordEncoder) {
      this.usersRepository = usersRepository;
      this.bCryptPasswordEncoder = bCryptPasswordEncoder;
   }


   @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.findById(id).get();
   }

   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.deleteById(id);
  }
}
```

Modificar el servicion InsertSampleDataService

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 anterior.

```
@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("99999977E", "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
```

4.4.5 Actualizar el Controlador UsersController

Primero inyectamos el servicio **SecurityService** y luego creamos los métodos ***signup() y login()*** para la registro y autenticación de usuarios, respectivamente. Hay que destacar que en el método **signup(),** se asignará automáticamente el rol de alumno al usuario que crea una cuenta en la aplicación.

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

```
@Controller
public class UsersController {
  private final UsersService usersService;
  private final SecurityService securityService;

 public UsersController(UsersService usersService, SecurityService securityService) {
  this.usersService = usersService;
  this.securityService = securityService;
}
```

A continuación, incluimos la implementación de las funciones: **signup, login y home:**

- **/signup** crea un nuevo usuario con el rol STUDENT, lo identifica automáticamente y redirige la navegación a home.
- **login** y **home** muestran únicamente sus vistas correspondientes.

```
@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() {
   return "login";
}

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

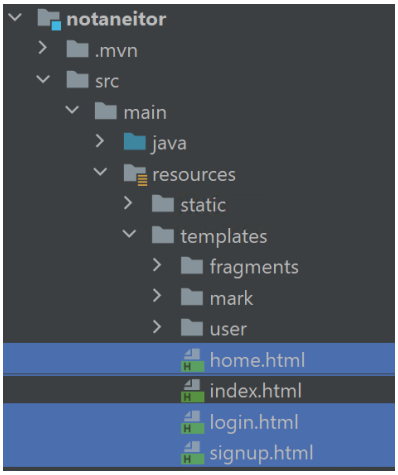
4.4.6 Configuración del adaptador de seguridad

El adaptador nos permite especificar qué recursos de la aplicación son, cuales no y bajo qué roles se accede a los mismos. En la clase **WebSecurityConfig** definimos el método **configure**.

```
....
import org.springframework.security.config.annotation.web.builders.HttpSecurity;

@Configuration
@EnableWebSecurity
public class WebSecurityConfig extends WebSecurityConfigurerAdapter {
....

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

4.5 Definir y actualizar vistas

Dentro de la carpeta templates vamos a crear las siguientes vistas: ***home.html, login.html y signup.html*** .



4.5.1 Vista signup.html

Creamos 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" xmlns="http://www.w3.org/1999/xhtml" xmlns:th="http://thymeleaf.org">
<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" id="dni" 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" id="name" 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" id="lastName" name="lastName"
          placeholder="Ejemplo: Peréz 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" id="password" name="password"
              placeholder="Entre el Password" />
        </div>
      </div>
      <div class="form-group" >
        <label class="control-label col-sm-2" id="passwordConfirm" 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 **templates/login.html** para la autenticación de los usuarios.

Es importante que los atributos tengan los valores: **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).

```
<!DOCTYPE html>
<html lang="en" xmlns="http://www.w3.org/1999/xhtml" xmlns:th="http://thymeleaf.org">
<head th:replace="fragments/head"/>
<body>

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

<div class="container">
  <h2>Identifícate</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" id="username"  name="username"
            placeholder="Ejemplo: profSDI" required="true" />
      </div>
    </div>
    <div class="form-group">
      <label class="control-label col-sm-2" for="username">Password:</label>
      <div class="col-sm-10">
        <input type="password" class="form-control" id="password" 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>
```

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.

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

Creamos la página **templates/home.html**, que será la primera página que visualicen los usuarios logueados.

```
<!DOCTYPE html>
<html lang="en" xmlns="http://www.w3.org/1999/xhtml" xmlns:th="http://thymeleaf.org">
<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>
```

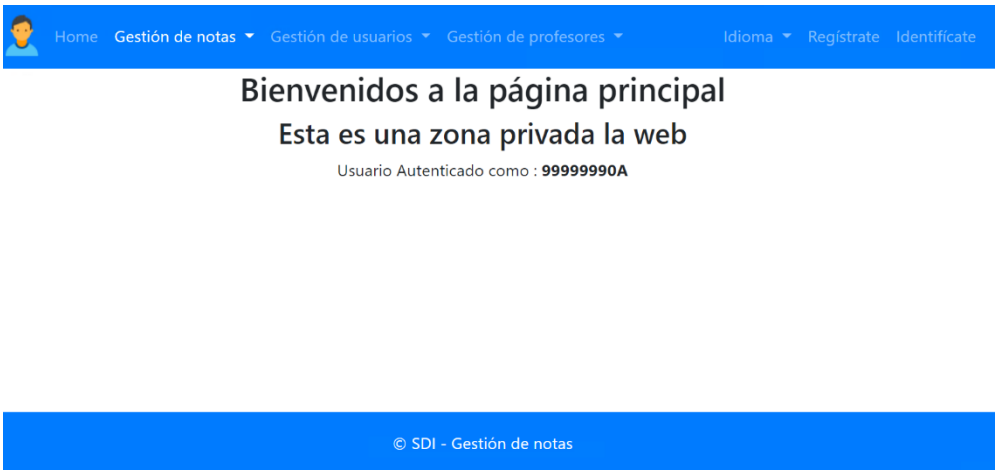
```
    Usuario Autenticado como :
    <b th:inline="text"> [[${#httpServletRequest.remoteUser}]] </b>
  </p>
</div>

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

</body>
</html>
```

Nota: En este punto, tenemos un problema de seguridad ya que todos los usuarios tienen acceso a gestionar usuarios, notas, … ***esto se resolverá más adelante.***

Probar la nueva funcionalidad

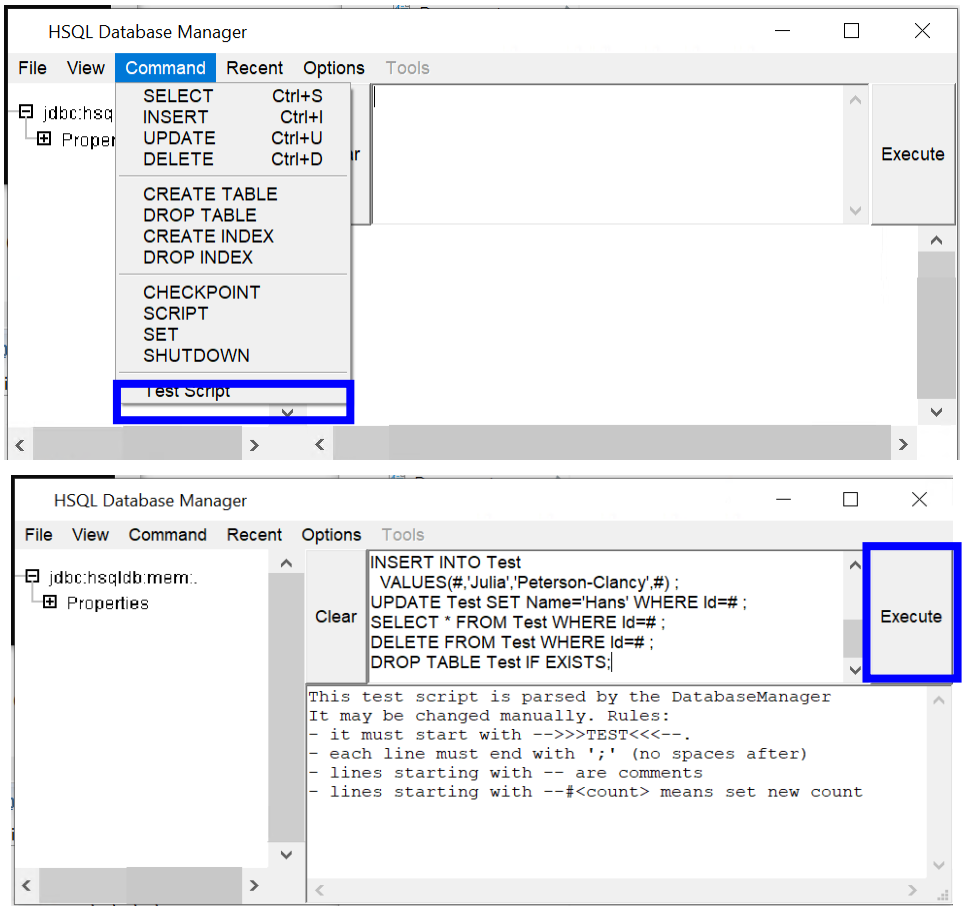
Ejecutamos la aplicación y probamos el sistema de identificación y autenticación, aquí tenemos algunos de los datos de prueba:

DNI: 99999990A password: 123456 ó DNI: 99999988F password: 123456



Nota: Si nos da algún error al crear la base de datos, por ejemplo: "…Caused by: java.sql.SQLSyntaxErrorException: usuario no tiene privilegios suficientes o objeto no encontrado: PUBLIC.MARK .." entonces desde HSQLDB manager ejecutamos el comando Test Script.

En la carpeta donde tenemos instalado el HSQLDB arrancamos el server Manager hsqldb-2.4.1\hsqldb\bin\runManager.bat y ejecutamos el script, tal como se muestra en las siguientes imágenes:



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.html** haciendo que procese un objeto **markList** que 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 :
    <b th:inline="text"> [[${#httpServletRequest.remoteUser}]] </b>
  </p>
  <p>Notas del usuario</p>
  <div class="table-responsive">
    <table class="table table-hover" th:fragment="marksTable" id="marksTable">
      <thead>
      <tr>
        <th scope="col">id</th>
        <th scope="col">Descripción</th>
```

```
            <th scope="col">Puntuación</th>
            <th scope="col"></th>
            <th scope="col"></th>
            <th scope="col"></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**,

- Obtenemos 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 autentificado: 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 = usersService.getUserByDni(dni);
   model.addAttribute("markList", activeUser.getMarks());
   return "home";
}
```

Nota: tenemos que importar los paquetes correspondientes: org.springframework.security.core.Authentication y org.springframework.security.core.context.SecurityContextHolder

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

Nota: Incluir el siguiente Commit Message ->

"SDI-IDGIT-3.4-Autenticación y control de acceso."

OJO: sustituir IDGIT por tu número asignado (p.e. 2324-101):

"SDI-2324-101-3.4-Autenticación y control de acceso."

(*No olvides incluir los guiones y NO incluyas BLANCOS al pricinpio o al final de la oración*)

5 Validación de datos en el servidor

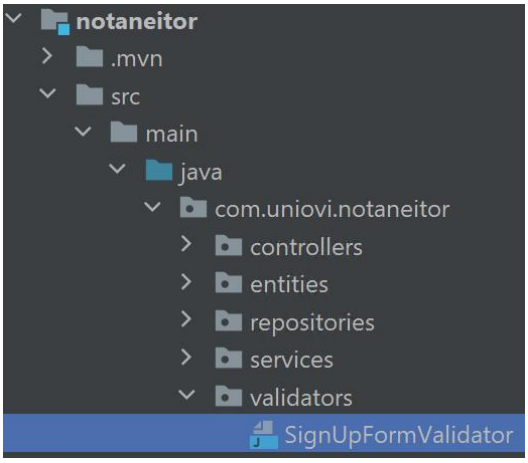
Cuando desarrollamos una aplicación web, es muy importante validar los datos de entrada que son enviados por los usuarios del sistema, para evitar errores y potenciales fallos de seguridad.

Las validaciones se pueden realizar en cliente mediante un lenguaje de script (ej., Javascript) o/y en el servidor. ***Para evitar problemas de seguridad lo recomendable es validar en ambos lados (cliente y 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.

Para validar el formulario de registro de usuarios del lado del servidor realizaremos los siguientes pasos:

Crear el validador
Creamos el paquete ***com.uniovi.notaneitor.validators,*** y luego la clase **SignUpFormValidator** que implemente la intefaz ***org.springframework.validation***, el primer formulario que vamos a validar es el formulario de registro de un nuevo usuario, presente en **signup.html**



Incluimos la validación de todos estos campos en la clase **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 relaccionado 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.

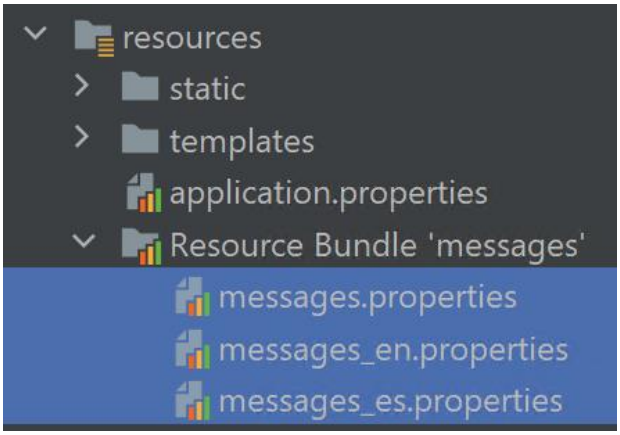
```
package com.uniovi.notaneitor.validators;
import com.uniovi.notaneitor.entities.User;
import com.uniovi.notaneitor.services.UsersService;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Component;
import org.springframework.validation.*;

@Component
public class SignUpFormValidator implements Validator {

    private final UsersService usersService;

    public SignUpFormValidator(UsersService usersService) {
        this.usersService = usersService;
    }

    @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 (usersService.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");}
    }
}
```

DNI existente: vamos a necesitar una función que nos permita buscar un usuario por Dni, para ver si este está duplicado. Accedemos a **UserService** y **UsersRepository** y si no está definido añadimos el método **findByDni** (no tenemos que implementar la función si esta lleva el nombre findBy<atributo>, se implementa de forma automática en el CrudRepository).

Finalmente, revisamos en la página **signup.html** que los nombres de los campos del formulario sean los siguientes: **dni, name, lastName, password, passwordConfirm**

Definir los mensajes de validación

A continuación, definimos todos los mensajes de validación en los ficheros **messages_en.properties** para inglés y **messages.properties** y **messages_es.properties** para Español.



Añadimos las siguientes líneas al fichero **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.
```

Añadimos las siguientes líneas a **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=Este DNI ya existe.
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.
```

Actualizamos la pagina signup.html

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 comprueba los errores de su campo especifico, **el dni, name, username, etc.**

```
<!DOCTYPE html SYSTEM "http://www.thymeleaf.org/dtd/xhtml1-strict-thymeleaf-spring4-4.dtd">
<html lang="en" xmlns="http://www.w3.org/1999/xhtml" xmlns:th="http://www.thymeleaf.org">
<head th:replace="fragments/head">
  <title></title>
</head>
<body>

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

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

```
    </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" id="passwordConfirm" name="passwordConfirm"
          placeholder="Repita el Password"/>
     <span class="text-danger" th:if="${#fields.hasErrors('passwordConfirm')}"
          th:errors="*{passwordConfirm}"></span>
    </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"></footer>
</body>
</html>
```

Modificar el controlador de usuarios

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

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

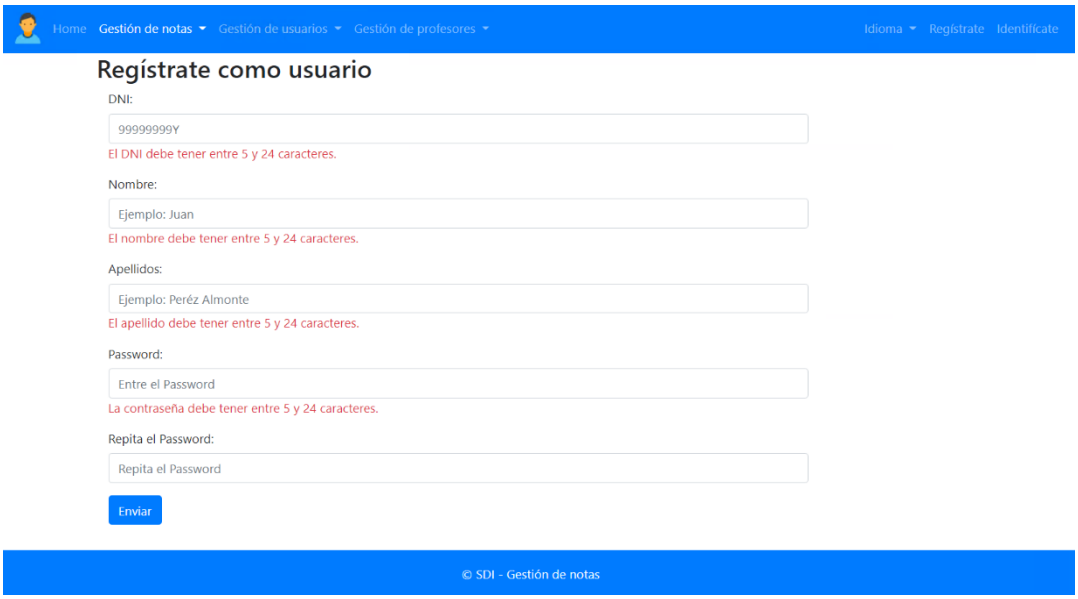
La parte que realmente realiza la validación será **POST/signup,** es la que recibe los datos del formulario y debe validarlos. En primer lugar, inyectamos el Componente **SignUpFormValidator**

```
import com.uniovi.notaneitor.validators.SignUpFormValidator;

@Controller
public class UsersController {
    private final SignUpFormValidator signUpFormValidator;
    public UsersController(UsersService usersService, SecurityService securityService, SignUpFormValidator
signUpFormValidator) {
     this.usersService = usersService;
     this.securityService = securityService;
    this.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) {
   signUpFormValidator.validate(user, result);
   if (result.hasErrors()) {
      return "signup";
   }

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

Finalmente, desplegamos la aplicación e intentamos registrarnos con datos erroneos para verificar el funcionamento de la validación.



Nota: Incluir el siguiente Commit Message ->

"SDI-IDGIT-3.5-Validación de datos en el servidor."

OJO: sustituir IDGIT por tu número asignado (p.e. 2324-101):

"SDI-2324-101-3.5-Validación de datos en el servidor."

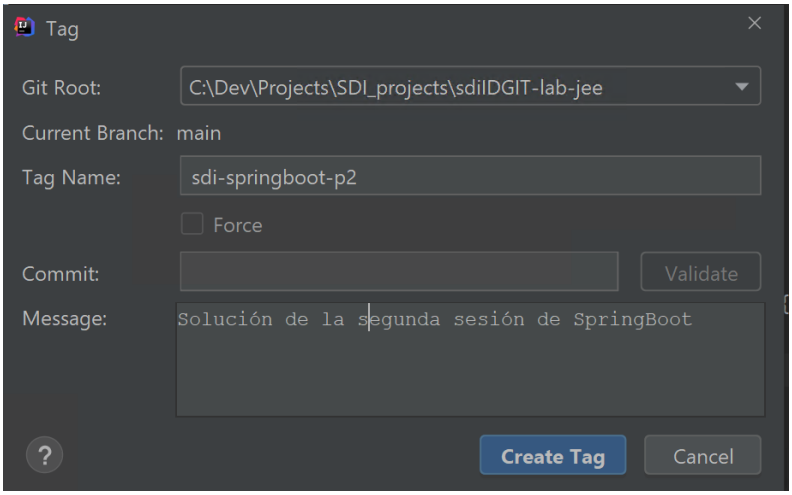
(*No olvides incluir los guiones y NO incluyas BLANCOS al pricinpio o al final de la oración*)

5.1 Etiquetar proyecto en GitHub

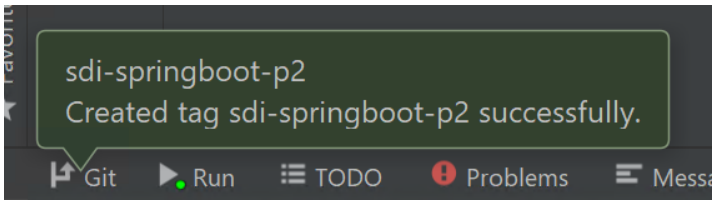
> **Nota:** Etiquetar el proyecto en este punto con la siguiente etiqueta -> **sdi-springboot-p2.**

Para crear una **Release** en un proyecto que está almacenado en un repositorio de control de versiones lo común es utilizar etiquetas (Tags), que nos permitirá marcar el avance de un proyecto en un punto dado (una funcionalidad, una versión del proyecto, etc). Para crear una etiqueta en el proyecto en GitHub lo primero que vamos a hacer es hacer click derecho sobre el proyecto en Intellij IDEA e ir a la opción de **Git |New Tag** que nos llevará hasta la ventana de creación de etiquetas(Tag).

En la siguiente ventana indicamos el nombre de la etiqueta(**sdi-springboot-p2)** y un mensaje que indique la funcionalidad que se está etiquetado en este punto, luego pulsamos el botón **Create Tag.**

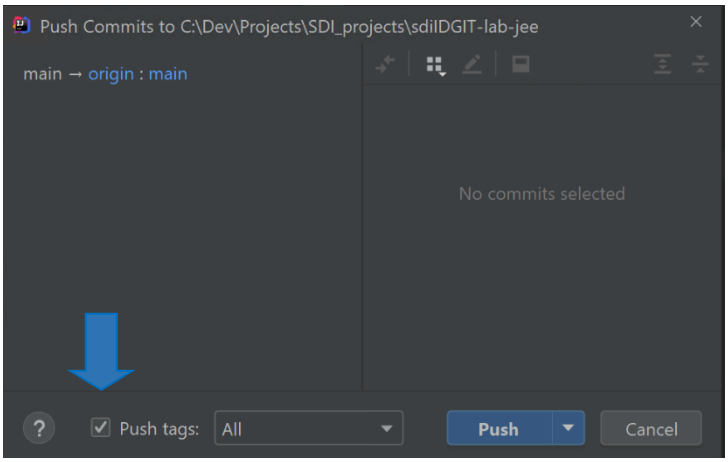


Nos saldrá un mensaje en el Intellij IDEA en la parte inferior izquierda, tal como se muestra en la siguiente imagen:

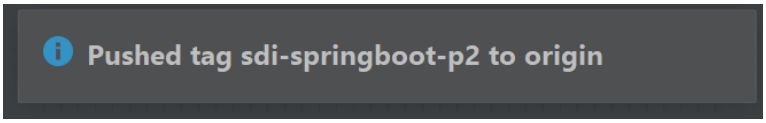


Con el paso anterior hemos creado la etiqueta en el repositorio local. Ahora tenemos que subir esa etiqueta al repositorio remoto (GitHub). Para subir una etiqueta al repositorio de Github hacemos click derecho sobre el proyecto e ir a la opción de **Git |Push**.

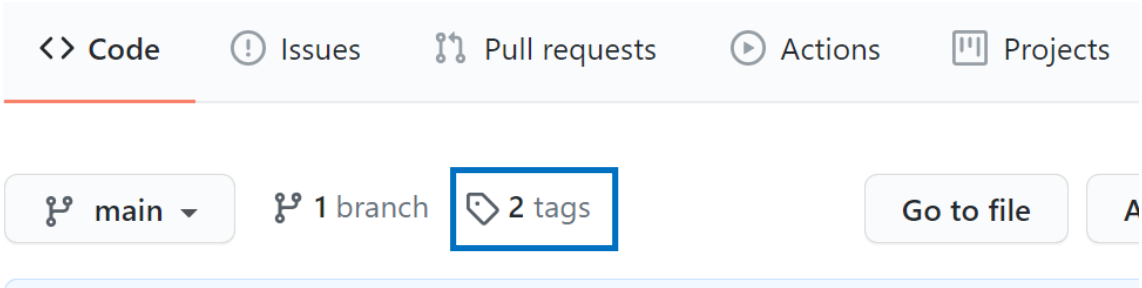
Una vez en la siguiente ventana seleccionamos la opción **Push tags** y pulsamos el botón Push.



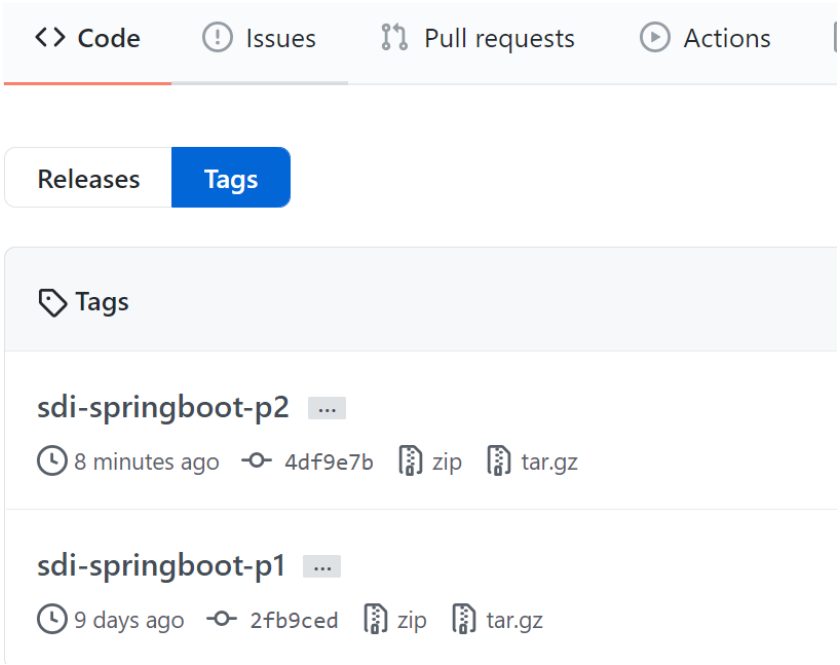
Al finalizar Intellij nos mostrará un mensaje en la parte inferior derecha con el resultado del push, tal como se muestra en la siguiente imagen:



Una vez subida las etiquetas al repositorio, podemos verificar dichas etiquetas en nuestra cuenta y repositorio de GitHub. **Vamos al repositorio correspondiente y pulsamos en el enlace de tags**.



Si se ha realizado la operación correctamente, debería ser visible la etiqueta creada anteriormente:



5.2 Resultado esperado en el repositorio de GitHub

Al revisar el repositorio de código en GitHub, los resultados de los commits realizados deberían ser parecidos a estos.

⇒ **SDI-2324-101-3.1-Thymeleaf, fragmentos.**
⇒ **SDI-2324-101-3.2-Internacionalización.**
⇒ **SDI-2324-101-3.3-Acceso a datos con entidades relacionadas.**
⇒ **SDI-2324-101-3.4-Autenticación y control de acceso.**
⇒ **SDI-2324-101-3.5-Validación de datos en el servidor.**