18-3-2018

Entregable 1

Sistemas distribuidos e internet

Iván González Mahagamage
UNIVERSISDAD DE OVIEDO

Autores: **Iván González Mahagamage**

Estudiante de Ingeniería Informática del Software en la Universidad de

Fecha: 18 de marzo de 2018

Versión: V1

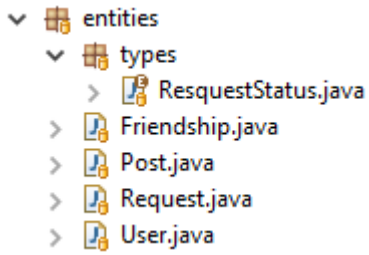
Contenido

Implementación

0. Aspectos comunes

0.1. Entidades de la aplicación.

Para la realización de la aplicación se ha diseñado una serie clases de objeto dentro del paquete “entities”.



Todos ellas están configuradas con la tecnología JPA para manejar la base de datos de manera automática. Las clases creadas son:

- User: simula los usuarios que están dentro de la aplicación.

```java
@Entity
@Table(name = "TUSERS")
public class User {
    @Id
    @GeneratedValue
    private long id;

    @Column(unique = true)
    @NotNull
    private String email;

    private String name;

    @Column(name = "surname")
    private String surName;

    @NotNull
    private String password;

    @Transient
    private String passwordConfirm;

    private String role;

    @OneToMany(mappedBy = "sender", cascade = CascadeType.ALL)
    private Set<Request> sentRequests = new HashSet<>();

    @OneToMany(mappedBy = "receiver", cascade = CascadeType.ALL)
    private Set<Request> receiveRequests = new HashSet<>();

    @OneToMany(mappedBy = "friend")
    private Set<Friendship> friends = new HashSet<>();

    @OneToMany(mappedBy = "user")
    private Set<Friendship> iAmFriendOf = new HashSet<>();

    @OneToMany(mappedBy = "user", cascade = CascadeType.ALL)
    private Set<Post> posts = new HashSet<>();
```

La información del usuario que se guarda es su email, es único y se usara para su identificación, nombre completo y contraseña. Además de las referencias a los otros objetos de la aplicación.

- Request: simula las peticiones de amistad que se envían entre sí.

```
@Entity
@Table(name = "TREQUEST")
public class Request {
    @Id
    @GeneratedValue
    private Long id;

    @ManyToOne
    private User sender;

    @ManyToOne
    private User receiver;

    @Enumerated(EnumType.STRING)
    private ResquestStatus status;
```

En este objeto además de guardar una referencia al usuario que manda la petición y el que la recibe, se guarda el estado de esta, que se indicara con enumerable. Este puede ser enviada, aceptada o bloqueada.

```
public enum ResquestStatus {
    SENT, ACCEPTED, BLOCKED
}
```

- Friendship: simula la amistad que existe entre los diferentes usuarios.

```
@Entity
@Table(name = "TFRIENDSHIP")
public class Friendship {

    @Id
    @GeneratedValue
    private Long id;

    @ManyToOne
    private User user;

    @ManyToOne
    private User friend;
```

Esta clase ha sido creada debido a los problemas entre el mapeador y la paginación de lista, que se explicará más adelante en el documento. La manera elegante de crear las amistades hubiera sido en el atributo “friends” de la clase “User” haber puesto la etiqueta “ManyToMany” pero a la hora de hacer la paginación de resultados la aplicación fallaba, cuando había más de una página. Por este motivo se optó por crear esta clase

- Post: publicaciones que realizan los usuarios.

```java
@Entity
@Table(name = "TPOSTS")
public class Post {
    @Id
    @GeneratedValue
    private Long id;

    @ManyToOne
    private User user;

    private String title, text, img;

    private Date date;
```

  En esta clase se guarda una referencia al usuario que ha creado la publicación, el título y el contenido se guardan directamente, pero se guarda la referencia a la imagen, no la misma en sí, pero esto ya se detallara más adelante.

También hay que decir que cada una de estas clases tiene un controlador, un servicio y repositorio.

0.2. Creación automática de datos.

Para tener datos para probar la aplicación se ha creado la clase “InsertSampleDataService” que tiene tres funciones:

- Crea un usuario Admin: ivangonzalezmahagamage@gmail.com.
- Crea un usuario User: igm1990@hotmail.com
- Genera el número de usuarios aleatorios que le indicamos.

A continuación, añado un fragmento de esta clase en la que se muestra como realiza estas opciones en el “@PostContruct”

```java
@PostConstruct
public void init() {
    // inicializar(50);
}

protected void inicializar(int limite) {
    user1 = new User("ivangonzalezmahagamage@gmail.com", "Iván",
            "González Mahagamage");
    user1.setPassword("123456");
    user1.setRole(rolesService.getAdmin());
    Post post = new Post(user1, "Prueba1", "Contenido", "");
    user1.getPosts().add(post);
    usersService.add(user1);

    User user2 = new User("igm1990@hotmail.com", "Iván",
            "González Mahagamage");
    user2.setPassword("123456");
    user2.setRole(rolesService.getUser());
    usersService.add(user2);
    rellenarBaseDatos(limite);

}

protected void rellenarBaseDatos(int limite) {
    while (users.size() < limite) {
        createUser();
    }

    Iterator<User> a = users.iterator();
    while (a.hasNext()) {
        usersService.add(a.next());
    }
}
```

0.3. Creación de del log de actividad.

Como se indica en el enunciado, se creado una clase denominada “LogService” para crear un log de todos los sucesos que ocurren en la aplicación. Para ello se añadido la siguiente dependencia al archivo “pom.xml”.

```xml
<dependency>
    <groupId>org.slf4j</groupId>
    <artifactId>slf4j-api</artifactId>
</dependency>
```

También en el fichero “aplicaction.properties” indicamos el fichero donde se guardará el log.

```
server.port=8090
spring.datasource.url=jdbc:hsqldb:hsql://localhost:9001
spring.datasource.username=SA
spring.datasource.password=
spring.datasource.driver-class-name=org.hsqldb.jdbcDriver
spring.jpa.hibernate.ddl-auto=validate
spring.messages.basename=messages/messages
logging.level.org.springframework.web=ERROR
logging.level.org.hibernate=ERROR
logging.file=log/log.log
spring.http.multipart.max-file-size=30MB
spring.http.multipart.max-request-size=30MB
```

La implementación de esta clase es la siguiente:

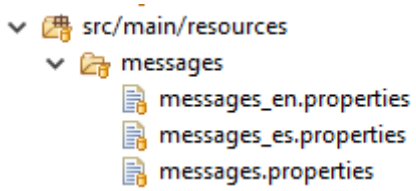
```
package com.uniovi.services;

import org.slf4j.Logger;
import org.slf4j.LoggerFactory;

import com.uniovi.services.util.CreateFolder;

public class LogService {

    private Logger log;

    public LogService(Object object) {
        CreateFolder.createFolder("/log");
        log = LoggerFactory.getLogger(object.getClass());
    }

    public void info(String message) {
        log.info(message);
    }

    public void error(String message) {
        log.error(message);
    }

    public void debug(String message) {
        log.debug(message);
    }

}
```

Se optado por esta manera para hacerla dinámica, únicamente tenemos que instanciarla en las clases que queremos registrar su actividad pasándole como parámetro “this” en su constructor. Un ejemplo:

```
@Controller
public class HomeController {

    private LogService logService = new LogService(this);

    @RequestMapping("/")
    public String index() {
        logService.info("Usuario a entrado en la aplicación");
        return "index";
    }

}
```

0.4. Proceso de internacionalización.

Para el proceso de internacionalización se han creado los ficheros .messages como se indicó en clase únicamente con una ligera variación, su localización. En mi caso, los he colocado dentro de un directorio denominado “messages” dentro del directorio “resources”

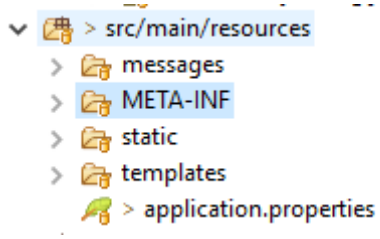


Para que la aplicación los reconozca, he tenido que hacer la siguiente modificación en el archivo “aplicaction.properties”.

```
server.port=8090
spring.datasource.url=jdbc:hsqldb:hsql://localhost:9001
spring.datasource.username=SA
spring.datasource.password=
spring.datasource.driver-class-name=org.hsqldb.jdbcDriver
spring.jpa.hibernate.ddl-auto=validate
spring.messages.basename=messages/messages
logging.level.org.springframework.web=ERROR
logging.level.org.hibernate=ERROR
logging.file=log/log.log
spring.http.multipart.max-file-size=30MB
spring.http.multipart.max-request-size=30MB
```

0.5. Externacionalización de las consultas.

Este apartado no se ha dado en clase pero yo he decido externalizar las consultas tal como nos enseñaron en la asignatura de “Repositorios de la Información”. Para ello he creado un directorio denominado “META-INF” dentro del directorio “resource”.



Dentro de este he creado un fichero denominado “jpa-named-queries.properties”, en el cual he añadido las diferente consultas que se usan en la aplicación. Para que esta funcione correctamente, se debe seguir la siguiente estructura:

- Nombre de la clase que devuelve la consulta.
- Nombre del método dentro del repositorio que referencia la consulta.
- Símbolo “=”
- Consulta

```
User.searchByEmailAndNameAndSurname=SELECT u FROM User u WHERE (LOWER(u.email) LIKE LOWER(?1) OR LOWER(u.name) L
User.findAllList=SELECT u FROM User u WHERE u.id != ?1 ORDER BY u.surName ASC
User.findAllByRequestReceiverId=SELECT r.sender FROM Request r WHERE r.status = 'SENT' AND r.receiver.id = ?1
User.findAllFriendsById=SELECT u.friend FROM Friendship u WHERE u.user.id = ?1
Request.findAllSentById=SELECT r FROM Request r WHERE r.sender.id = ?1
Request.findBySenderIdAndReceiverId=SELECT r FROM Request r WHERE r.sender.id = ?1 AND r.receiver.id = ?2
Post.findAll=SELECT p FROM Post where p.user.id = ?1
Friendship.searchByUsers=SELECT f FROM Friendship f WHERE f.user = ?1 AND f.friend = ?2
```

0.6. Roles de la aplicación

He establecido dos roles para la aplicación “ROLE_USER” para usuario sin privilegios especiales y “ROLE_ADMIN” para usuario con privilegios de administrador. Esto se realizado mediante la creación de la clase “RolesService”.

```
@Service
public class RolesService {

    public String getUser() {
        return "ROLE_PUBLIC";
    }

    public String getAdmin() {
        return "ROLE_ADMIN";
    }
}
```

0.7. Definición de permisos

Para esta aplicación hemos definidos los siguientes permisos en la clase “WebSecurityConfig”.

```
@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() // peticiones autorizadas
                .antMatchers("/css/**", "/img/**", "/script/**", "/",
                        "/signup", "/login", "/admin/login")
                .permitAll()
                // Permite a todos los usuarios
                .antMatchers("/admin/**").hasAuthority("ROLE_ADMIN")
                .anyRequest().authenticated()
                // pagina de autentificacion por defecto
                .and().formLogin().loginPage("/login").permitAll()
                // Si se loguea bien
                .defaultSuccessUrl("/home").and().logout().permitAll();
    }

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

Definimos que los css, imágenes, scripts, las pagina de inicio, registrase, login, y login del administrador pueda acceder cualquier usuario. También definimos que las url que sean de administrador solo pueden acceder los usuarios que tengan el rol “ADMIN_USER”

0.8. Utilidades

He reutilizado una clase denominada “createFolder” para crear directorios de manera dinámica para el log de la aplicación y el guardado de las imágenes subidas por los usuarios.

```
/**
 * Clase encargada de crear las carpetas necesarias para nuestra aplicación.
 *
 * @author Adrián García Lumbreras
 * @author Iván González Mahagamage
 *
 */
public class CreateFolder {
    /**
     * Crea una carpeta por código si esta no existe previamente.
     *
     * @param url
     */
    public static void createFolder(String url) {
        File file = new File(url);
        if (!file.exists()) {
            file.mkdir();
        }
    }
}
```

1. Público: registrarse como usuario

Primero se ha creado una vista para este propósito denominada “signup.html”. De esta vista solo voy destacar ciertos aspectos ya que es demasiado código para ponerlo en este documento.

- Al formulario le insertamos un object para que si el registro no es válido, guarde los datos que ha introducido el usuario a excepción de las contraseñas para evitar que los tenga que volver a introducir.

```
<form method="post" action="/signup" th:object="${user}">
```

- A parte de introducir la internacionalización en todas las vistas, he creado un archivo en JavaScript para incluir validaciones por el lado del cliente además de incluir el atributo placeholder.

```
<div class="form-group">
    <label th:text="#{signup.name}">Nombre:</label>
    <input class="form-control" type="text" name="name" placeholder="Iván" required="required"
           onkeyup="check.checkInputs()" th:attr="title=#{signup.name.title}, value=${user.name}"
           pattern="[a-zA-ZñÑáéíóúÁÉÍÓÚ]{2,}" />
    <div th:if="${#fields.hasErrors('name')}" class="alert alert-dismissible alert-danger">
        <button type="button" class="close" data-dismiss="alert">&times;</button>
        <strong th:errors="*{name}">Oh snap!</strong>
    </div>
</div>
```

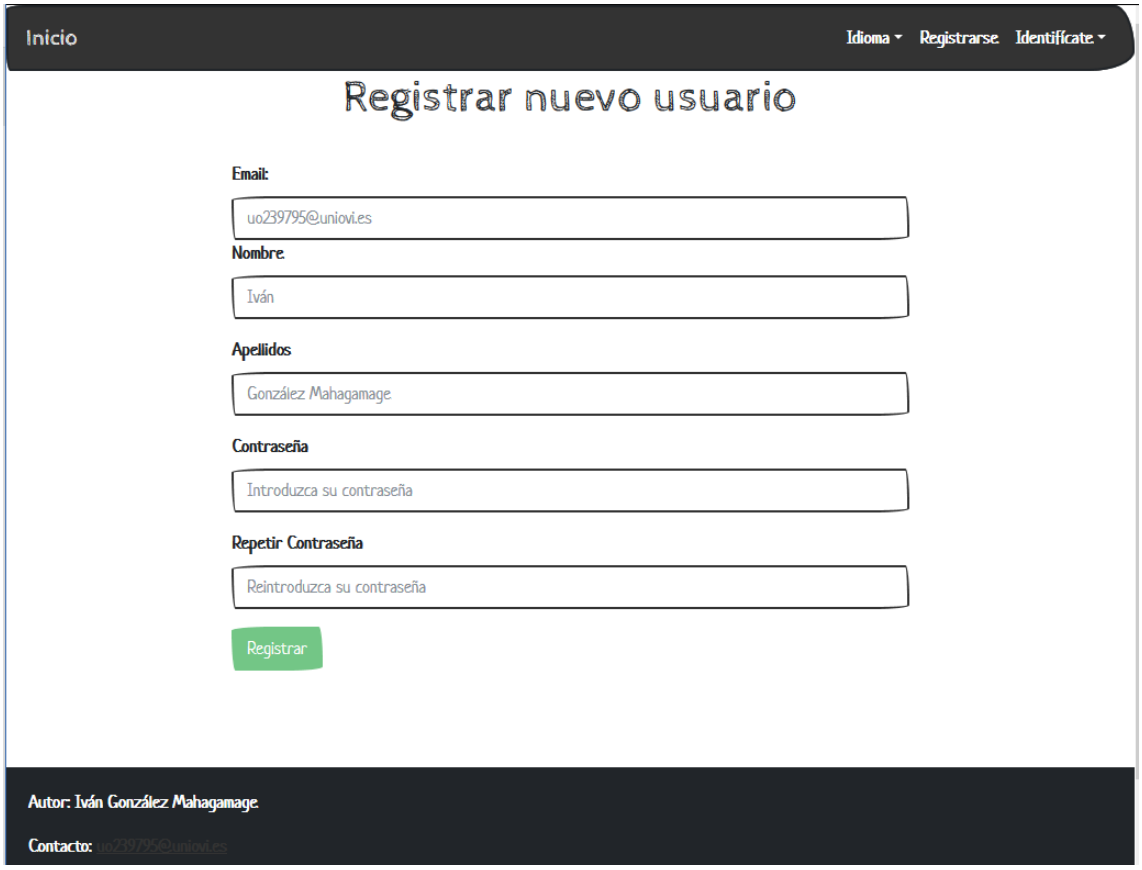
- El archivo JavaScript denominado anterior es “checkForm.js”.”

```
<body>
    <script src="/script/checkForm.js"></script>
```

Comprueba que las dos contraseñas que introduce el usuario son iguales y hasta que el usuario no introduce todos los campos del formulario desactiva el botón de “Enviar.

```
"use strict";

class Check {
    constructor() {}

     checkPasswords() {
        var password = $('input[name=password]').val();
        var repassword = $('input[name=passwordConfirm]').val();
        if (password !== repassword) {
            alert("Error: las contraseñas no coinciden");
            return false;
        }
        return true;
    }

    checkInputs() {
        $('input[type=submit]').prop('disabled', false);
        var inputs = $('input');
        for (var i = 0; i < inputs.length; i++) {
            var value = inputs[i].value;
            if (inputs[i].value === "") {
                $('input[type=submit]').prop('disabled', true);
                break;
            }
        }
    }
}

var check = new Check();
```

El resultado final de estos archivos es:



Para que la vista se muestre correctamente debemos incluir en UserController los siguientes métodos:

- Un método para recoja la petición Get para mostrar la vista.

```
@GetMapping("/signup")
public String signUp(Model model) {
    logService.info("Usuario se intenta registrar");
    model.addAttribute("user", new User());
    return "signup";
}
```

- Un método para recoja la petición Post y procese el formulario.

```
@PostMapping("/signup")
public String signupPost(@Validated User user, BindingResult result,
        Model model) {
    signUpFormValidator.validate(user, result);
    if (result.hasErrors()) {
        logService.error("Usuario introdujo mal los datos");
        return "signup";
    }
    logService.info("Usuario se ha registrado correctamente como "
            + user.getEmail());
    user.setRole(rolesService.getUser());
    usersService.add(user);
    securityService.autoLogin(user.getEmail(), user.getPasswordConfirm());
    return "redirect:home";
}
```

En este método incluimos una llamada a un validador para comprobarlo en lado del servidor.

```
@Component
public class SignUpFormValidator implements Validator {
    @Autowired
    private 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, "email",
                "Error.empty");
        if (usersService.getUserByEmail(user.getEmail()) != null) {
            errors.rejectValue("email", "Error.signup.email.duplicate");
        }
        if (user.getName().length() < 2) {
            errors.rejectValue("name", "Error.signup.name.length");
        }
        if (user.getSurName().length() < 2) {
            errors.rejectValue("surName", "Error.signup.lastName.length");
        }
        if (user.getPassword().length() < 5) {
            errors.rejectValue("password", "Error.signup.password.length");
        }
        if (user.getPasswordConfirm().length() < 5) {
            errors.rejectValue("passwordConfirm",
                    "Error.signup.password.length");
        }
        if (!user.getPasswordConfirm().equals(user.getPassword())) {
            errors.rejectValue("passwordConfirm",
                    "Error.signup.passwordConfirm.coincidence");
        }
    }
}
```

Si supera las comprobaciones se le asigna el rol “ROLE_USER” y se llama a UserService para que lo añada a la base de datos y encripte su contraseña.

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

Una vez finalizado este proceso, la aplicación loguea automáticamente al usuario para que comience a usar la aplicación. Para ello también debemos implementar la clase “SecurityService”.

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

    @Autowired
    private UserDetailsService userDetailsService;
    private static final Logger logger = LoggerFactory
            .getLogger(SecurityService.class);

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

    public void autoLogin(String email, String password) {
        UserDetails userDetails = userDetailsService.loadUserByUsername(email);
        UsernamePasswordAuthenticationToken aToken;
        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!", email));
        }
    }
}
```

A su vez, esta clase llama a “UserDetailsServiceImpl”

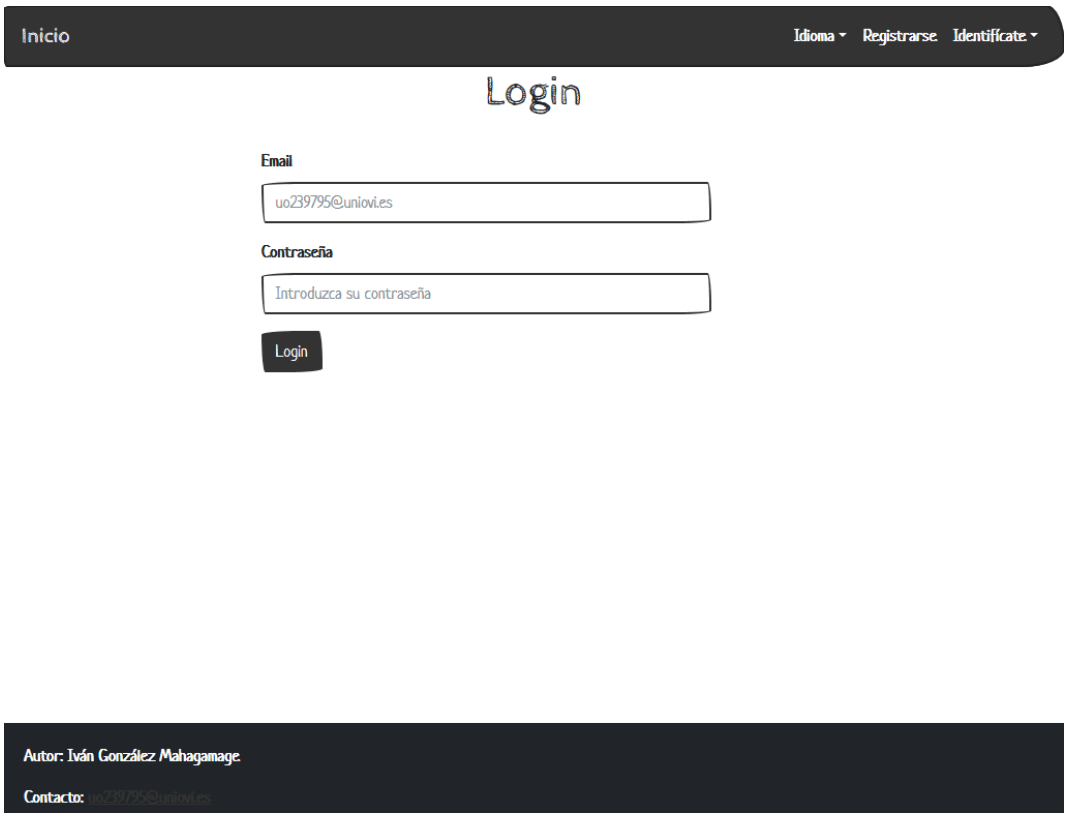
```java
@Service
public class UserDetailsServiceImpl implements UserDetailsService {
    @Autowired
    private UsersRepository usersRepository;

    @Override
    public UserDetails loadUserByUsername(String email)
            throws UsernameNotFoundException {
        com.uniovi.entities.User user = usersRepository.findByEmail(email);
        Set<GrantedAuthority> grantedAuthorities = new HashSet<>();
        grantedAuthorities.add(new SimpleGrantedAuthority(user.getRole()));
        return new User(user.getEmail(), user.getPassword(),
                grantedAuthorities);
    }

}
```

2. Público: iniciar sesión

Se ha implementado la vista “login.html” para que el usuario pueda iniciar sesión.

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

<body>
    <nav th:replace="fragments/nav" />
    <main class="container">
        <h1 th:text="#{login.title}">Idéntificate</h1>
        <form class="form-horizontal" method="post" action="/login">
            <fieldset>
                <div class="form-group">
                    <label class="control-label col-sm-2" for="username">Email</label>
                    <div class="col-sm-10">
                        <input type="email" class="form-control" name="username"
                               placeholder="uo239795@uniovi.es" required="required" th:attr="title=#{signup.email.title}" />
                    </div>
                </div>
                <div class="form-group">
                    <label class="control-label col-sm-2" for="password" th:text="#{signup.password}">Password:</label>
                    <div class="col-sm-10">
                        <input type="password" class="form-control" name="password"
                               th:attr="placeholder=#{signup.password.placeholder}, title=#{signup.password.placeholder}"
                               pattern=".{2,}" required="required" />
                    </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}" />
            </fieldset>
            <div th:if="${param.error}" class="alert alert-dismissible alert-danger">
                <button type="button" class="close" data-dismiss="alert">&times;</button>
                <strong th:text="#{login.error}">Oh snap!</strong>
            </div>
            <div th:if="${param.logout}" class="alert alert-dismissible alert-success">
                <button type="button" class="close" data-dismiss="alert">&times;</button>
                <strong th:text="#{login.logout}">Oh snap!</strong>
            </div>
        </form>
    </main>
    <footer th:replace="fragments/footer" />
</body>

</html>
```

El resultado es el siguiente:



Se declarado un nuevo método en “UserController” para mostrar esta vista.

```
@GetMapping("/login")
public String login() {
    logService.info("Usuario se intenta loggear");
    return "login";
}
```

Cabe destacar que como se usa Spring Security no hay que implementar la petición post de la vista.

3. Usuario registrado: listar todos los usuarios de la aplicación
4. Usuario registrado: buscar entre todos los usuarios de la aplicación
5. Usuario registrado: enviar una invitación de amistad a un usuario
6. Usuario registrado: listar las invitaciones de amistad recibidas
7. Usuario registrado: aceptar una invitación recibida
8. Usuario registrado: listar los usuarios amigos
9. Usuario registrado: crear una nueva publicación
10. Usuario registrado: listar mis publicaciones
11. Usuario registrado: listar las publicaciones de un usuario amigo
12. Usuario registrado: crear una publicación con una foto adjunta
13. Público: iniciar sesión como administrador
14. Consola de administración: listar todos los usuarios de la aplicación
15. Consola de administración: Consola de administración: eliminar usuario

Prueba Unitarias

1.1. Registro de Usuario con datos válidos.

1.2. Registro de Usuario con datos inválidos (repetición de contraseña invalida).

2.1. Inicio de sesión con datos válidos.

2.2. Inicio de sesión con datos inválidos (usuario no existente en la aplicación).

3.1. Acceso al listado de usuarios desde un usuario en sesión.

3.2. Intento de acceso con URL desde un usuario no identificado al listado de usuarios desde un usuario en sesión.

4.1. Realizar una búsqueda valida en el listado de usuarios desde un usuario en sesión.

4.2. Intento de acceso con URL a la búsqueda de usuarios desde un usuario no identificado.

5.1. Enviar una invitación de amistad a un usuario de forma valida.

5.2. Enviar una invitación de amistad a un usuario al que ya le habíamos invitado la invitación previamente.

6.1. Listar las invitaciones recibidas por un usuario.

7.1. Aceptar una invitación recibida.

8.1. Listar los amigos de un usuario, realizar la comprobación con una lista que al menos tenga un amigo.

9.1. Crear una publicación con datos válidos.

10.1. Acceso al listado de publicaciones desde un usuario en sesión.

11.1. Listar las publicaciones de un usuario amigo.

11.2. Utilizando un acceso vía URL tratar de listar las publicaciones de un usuario que no sea amigo del usuario identificado en sesión.

12.1. Crear una publicación con datos válidos y una foto adjunta.

12.2. Crear una publicación con datos válidos y sin una foto adjunta.

13.1. Inicio de sesión como administrador con datos válidos.

13.2. Inicio de sesión como administrador con datos inválidos.

14.1. Desde un usuario identificado en sesión como administrador listar a todos los usuarios de la aplicación.

15.1. Desde un usuario identificado en sesión como administrador eliminar un usuario existente en la aplicación.

15.2. Intento de acceso vía URL al borrado de un usuario existente en la aplicación.

Otros aspectos

1. Usuario registrado: bloquear invitaciones de amistad de un usuario