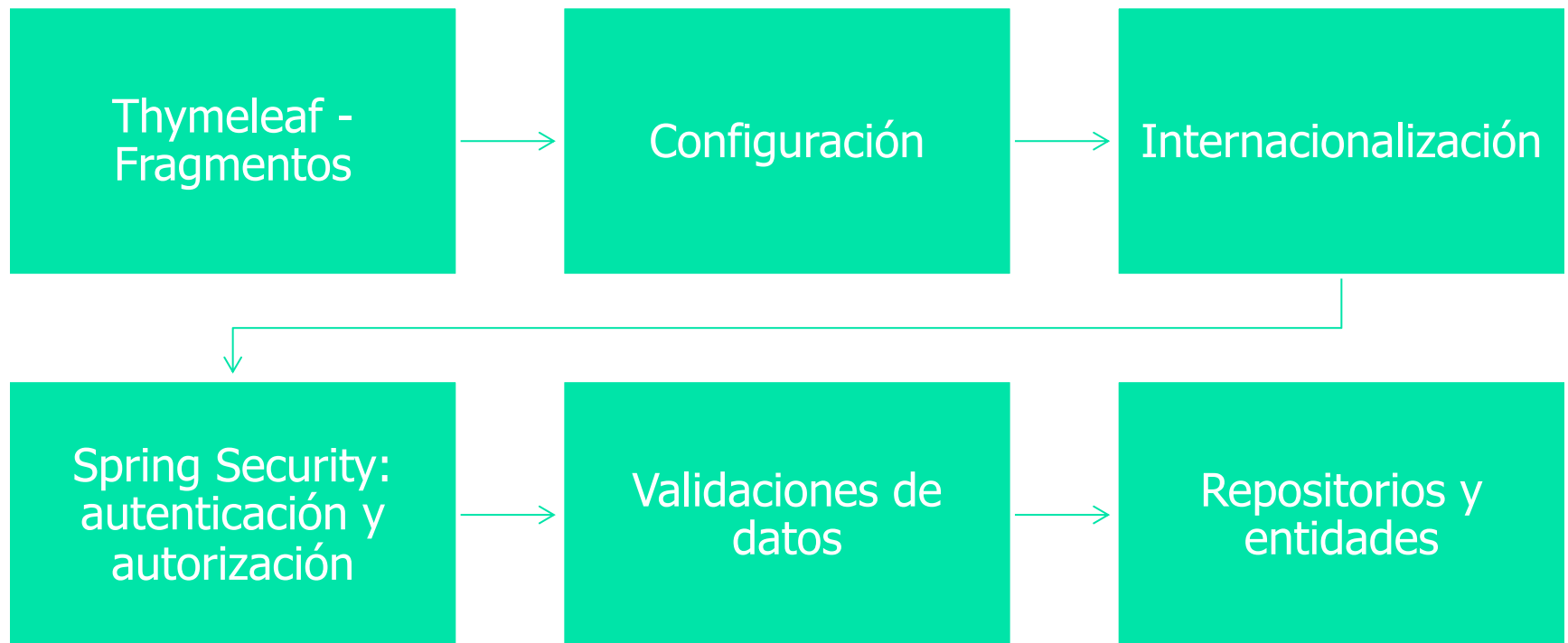




Sistemas Distribuidos e Internet

Tema 2
Spring Boot 2

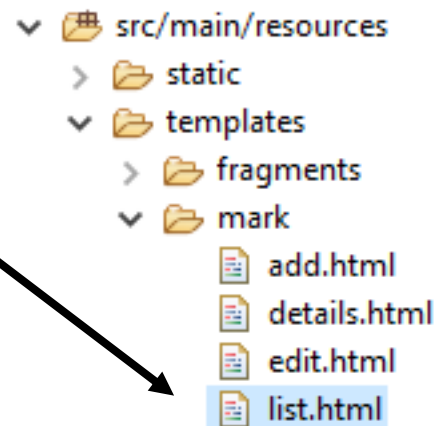
Contenidos



Fragmentos > Introducción

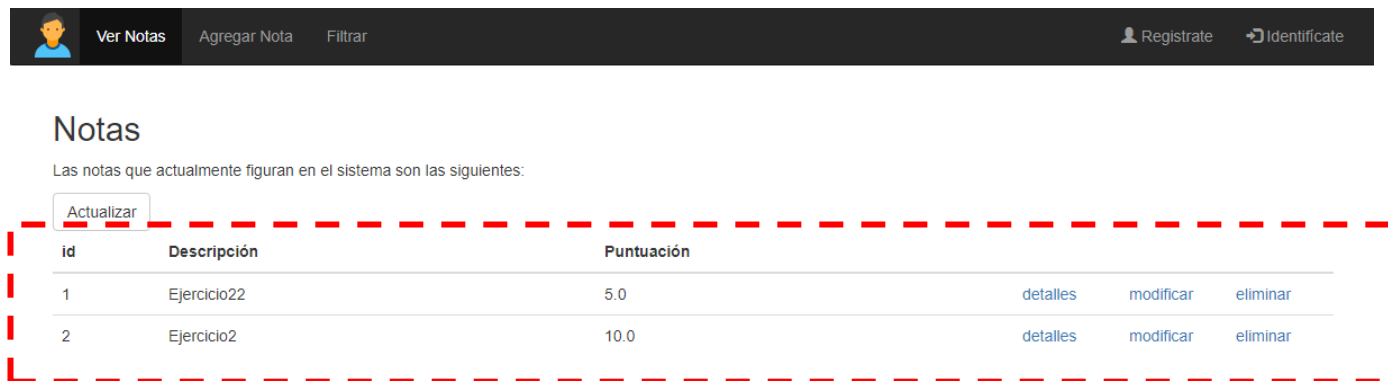
- El uso **por defecto** de Thymeleaft consiste en retornar una vista correspondiente a una plantilla
 - Ejemplo: plantilla **mark/list**
 - El navegador carga la nueva página

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



Fragmentos > Introducción

- Para ganar **fluidez y eficiencia** en ocasiones **no se retorna una plantilla/página completa**
- Una alternativa es retornar solo **una/varias partes fragmentos de la plantilla/página**
- Estos fragmentos se **sustituyen o insertan** en la página previa
 - Ej: ¿Cargamos toda la página para actualizar la lista de Notas?



Ver Notas Agregar Nota Filtrar Registrarse Identificarse

Notas

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

Actualizar

id	Descripción	Puntuación	
1	Ejercicio22	5.0	detalles modificar eliminar
2	Ejercicio2	10.0	detalles modificar eliminar

Fragmentos > th:fragment

- El atributo **th:fragment** delimita un fragmento en una plantilla
 - Bloque XML de código

```
<div class="form-group">
    <input name="searchText" type="text" class="form-control" size="50"
        placeholder="Buscar por descripción o nombre del alumno">
    </div>
    <button type="submit" class="btn btn-default">Buscar</button>
</form>
<div class="table-responsive">
    <table class="table table-hover" th:fragment="tableMarks" id="tableMarks">
        <thead>
            <tr>
                <th class="col-md-1">id</th>
                <th>Descripción</th>
                <th>Puntuación</th>
            </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>
            </tr>
        </tbody>
    </table>
</div>
```

Fragmento tableMarks

Fragmentos > th:fragment

- El número de **th:fragment** puede ser variable
 - Algunas plantillas no tienen ninguno, otras muchos
 - Depende del enfoque de desarrollo (eficiencia y dinamismo)
- Se suele dar una **id** al nodo padre del **th:fragment**
 - La id facilita referenciarlo desde javaScript

```
<table class="table table-hover" th:fragment="tableMarks" id="tableMarks">
```

Fragmentos > Controladores

- Los controladores referencian los fragments con:
 - **<ruta plantilla> :: <nombre fragmento>**
 - Ej: este controlador solo retorna el **fragmento tableMarks** de la vista

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

- La petición **/mark/list/update** :
 - Si accedemos a la URL se cargara solo el HTML del fragmento (vista incompleta de la web)
 - Se suelen invocar y cargar desde un **script cliente (JavaScript)**
 - El retorno(fragmento) **se inserta/sustituye en la página actual**

Fragmentos > Cliente

- Incluimos código de script (jQuery) en las vistas/plantillas para:
 1. Obtener el fragmento
 2. Incluirllo dentro de un elemento de la página actual
- Ej: la función **`$(<selector>).load(<url>)`** permite obtener y cargar el fragmento
 - Ej: carga el fragmento en el elemento con **`id="tableMarks"`**

```
<button type="button" id="updateButton">Actualizar</button>
<script>
    $( "#updateButton" ).click(function() {
        $("#tableMarks").load('/mark/list/update');
    });
</script>
<div class="table-responsive">
```


Configuración > Introducción

- Spring Boot incluye una configuración por defecto
- Puede ser modificada/ampliada
- La funcionalidad relativa a la configuración es **genérica**
 - No tiene que ver directamente con la lógica de negocio
 - Algunas funciones comunes de configuración son:
 - Configuración del sistema de seguridad (autenticación y autorización)
 - Configuración del sistema de paginación
 - Configuración del sistema de internacionalización
- Estas clases incluyen la anotación **@Configuration**
 - Los **@Configuration** son **componentes**
 - No se instancian manualmente, son registrados por **@ComponentScan**
- Suelen heredar de una **clase de configuración del framework**
 - Ej: **WebSecurityConfigurerAdapter** clase que define la configuración de seguridad

Configuración > @Configuration

- Las **clases base de configuración** del framework se pueden utilizar de diferentes formas, lo más común:
 - **Sobrescribiendo métodos** para personalizar el funcionamiento
 - **Utilizando métodos definidos en la clase de configuración**
- Ejemplo (parcial)
 - Configuración de seguridad, clase base **WebSecurityConfigureAdapter**
 - Sobrescribir **configure(HttpSecurity http)**

```
@Configuration
public class WebSecurityConfig extends WebSecurityConfigurerAdapter {

    @Override
    protected void configure(HttpSecurity http) throws Exception {
        . . .
        Usar el objeto HttpSecurity para configurar las autorizaciones
        . . .
    }
}
```

Configuración > @Bean

- En muchos casos las clases de configuración instancian objetos como Beans "básicos"
- Estos **objetos** definen:
 - Funcionalidad necesaria para la **propia configuración**
 - Funcionalidad **común** que será utilizada en otras partes de la aplicación
- Lo más común es que sean **clases del framework**
 - Ej: la clase **BCryptPasswordEncoder**, incluye métodos para codificar información con el algoritmo **BCrypt**
- Se declaran con la anotación **@Bean**
 - A diferencia de los componentes, se debe instanciar el objeto

```
@Configuration
public class WebSecurityConfig extends WebSecurityConfigurerAdapter {

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

    . . .
}
```

Configuración > @Bean

- Al estar en una clase **@Configuration** los Beans se registran al iniciar la aplicación

```
[org.springframework.context.annotation.internalConfigurationAnnotationProcessor,  
org.springframework.context.annotation.internalAutowiredAnnotationProcessor,  
...  
delegatingApplicationListener, webSecurityExpressionHandler, springSecurityFilterChain,  
privilegeEvaluator, autowiredWebSecurityConfigurersIgnoreParents,  
org.springframework.security.config.annotation.web.configuration.WebMvcSecurityConfiguration,  
requestDataValueProcessor, bCryptPasswordEncoder,  
...]
```

- Los **Beans** pueden ser inyectados en el código de la aplicación
 - Ej: inyección de bean

```
@Service  
public class UserService {  
  
    @Autowired  
    private UserRepository userRepository;  
    @Autowired  
    private BCryptPasswordEncoder bCryptPasswordEncoder;  
}
```

Internacionalización > Introducción

- Internacionalización acrónimo i18n
- Consisten en adaptar una aplicación a diferentes idiomas o regiones
- Se trata de traducir todos los **contenidos y estándares a los propios de un idioma o región**
- Puede afectar a **muchos aspectos** de la aplicación
 - *Traducciones de textos e imágenes <- Principal
 - Estándares y formatos (divisas, fechas, horas, formatos numéricos, etc.)
 - Ordenes alfabéticos
 - Contenidos localizados (diferentes para idiomas o regiones)
 - Símbolos, iconos / colores dependientes de culturas
 - Exigencias legales
 - Otros

Internacionalización > Configuración

- Incluir **internacionalización** requiere **ampliar la configuración** de la aplicación
- Se añade una clase de **@Configuration** que extienda de **WebMvcConfigurerAdapter**
- **WebMvcConfigurerAdapter** es una de las clases más genéricas de configuración
 - Agrupa muchas funcionalidades. EJ: agregar **interceptores** (método **addInterceptors**)
- Los **interceptores** pueden procesar peticiones antes de que lleguen al controlador
 - Algunos usos: labores de autorización, registro de acceso, procesar parámetros comunes a todas las URLs, etc.

Internacionalización > Configuración

- Implementación de **interceptores** :
 - Clase que hereda de un interceptor
 - Ej: interceptor básico **HandlerInterceptorAdapter**
 - Obtiene el valor del parámetro "límite"

```
public class MiInterceptor extends HandlerInterceptorAdapter {  
  
    @Override  
    public boolean preHandle(HttpServletRequest request,  
        HttpServletResponse response, Object object) throws Exception {  
  
        System.out.println("Interceptar la petición");  
        // Buscar el parámetro "limite"  
        Integer parametroLimite =  
            ServletRequestUtils.getIntParameter(request, "limite", 0);  
  
        return true;  
    }  
}
```

Internacionalización > Interceptor

■ Agregar **interceptores** :

- Se obtiene un **Bean** del interceptor
 - Ej, puede ser un componente, objeto instanciado como @Bean, etc.
- Se registra el Bean a través del método **addInterceptors()** de una configuración basada en la clase **WebMvcConfigurerAdapter**

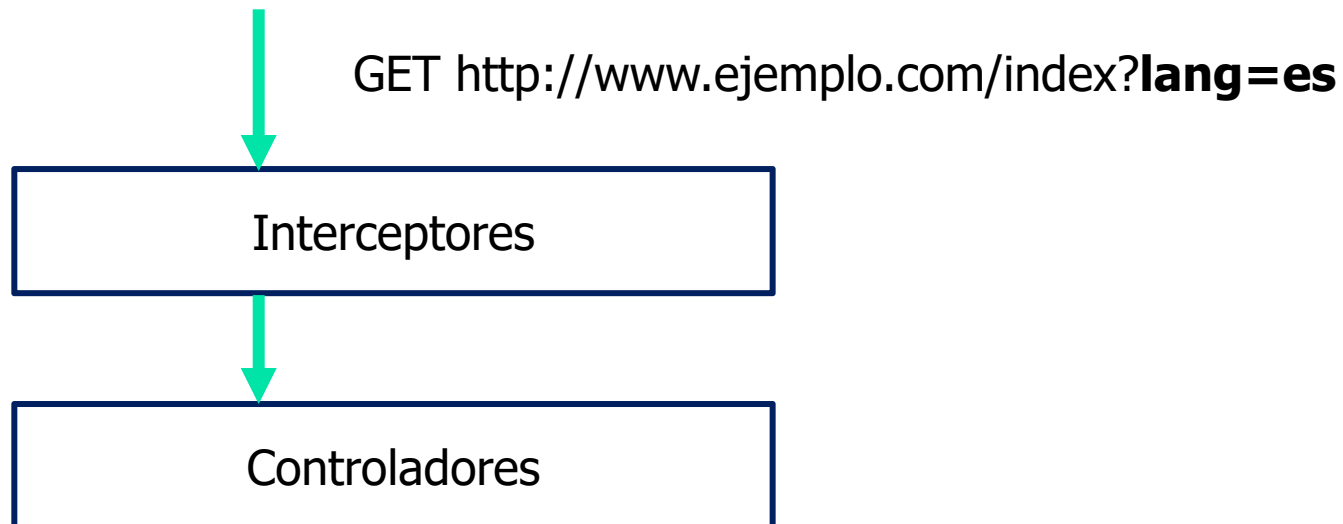
```
@Configuration
public class MiConfiguration extends WebMvcConfigurerAdapter{

    @Bean
    public MiInterceptor miInterceptor() {
        MiInterceptor miInterceptor = new MiInterceptor();
        return miInterceptor;
    }

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


Internacionalización > Interceptor

- **LocaleChangeInterceptor** es un interceptor implementado en el framework (hay muchas otras clases de interceptores)
 - Permite definir un parámetro para realizar cambios de “Localización”
 - Sí la petición contiene el parámetro se cambia la localización
 - Por ejemplo, parámetro **lang=<código de idioma>**
 - El interceptor esta activo sobre todas las URLs del sitio
 - No es necesario modificar los controladores



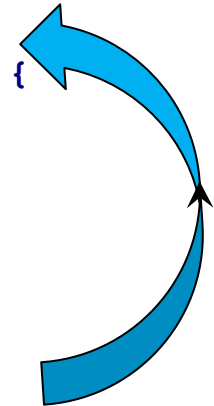
Internacionalización > Interceptor

■ Implementación

- Se crea un **Bean** con una instancia de **LocaleChangeInterceptor**
 - Se configura debidamente, indicando el nombre del parámetro del idioma
- Se sobrescribe el método **addInterceptors** y se registra el Bean Interceptor

```
@Configuration
public class mii18N extends WebMVCCConfigurer {
    @Bean
    public LocaleChangeInterceptor localeChangeInterceptor() {
        LocaleChangeInterceptor localeChangeInterceptor =
            new LocaleChangeInterceptor();
        localeChangeInterceptor.setParamName("lang");
        return localeChangeInterceptor;
    }

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




Internacionalización > LocaleResolver

- **LocaleResolver** objeto del framework permite hacer cambios automáticos de idioma (también especificar uno por defecto)
 - Se basa en las sesiones, cookies y cabeceras accept-language

▼ Request Headers

```
:authority: s0.2mdn.net
:method: GET
:path: /6644023/1512036500331/index.html
accept-encoding: gzip, deflate, br
accept-language: en,es-ES;q=0.9,es;q=0.8
```

Petición desde un Chrome
en idioma español



- Para habilitar la funcionalidad se debe registrar una instancia de **LocalResolver** como Bean
 - Instanciamos un objeto como Bean (@Bean)

```
@Configuration
public class CustomConfiguration extends WebMvcConfigurerAdapter{

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

Internacionalización > Mensajes

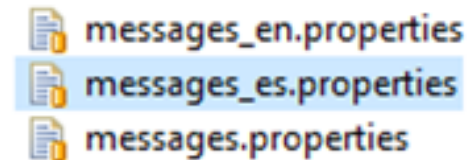
- Las cadenas de texto internacionalizadas se definen en **ficheros de propiedades**

- Cada cadena tiene una **clave** = **valor**

`welcome.message=Welcome to homepage`

- Se suelen usar **varios ficheros** de propiedades, **uno por cada localización**

- Idioma - https://es.wikipedia.org/wiki/ISO_639-1
- Países - https://es.wikipedia.org/wiki/ISO_3166-1



- Cada fichero define un **valor** para las cadenas

message_es `welcome.message=Bienvenidos a la página principal`

message_en `welcome.message=Welcome to homepage`

Internacionalización > Mensajes

- Desde **Thymeleaf** usamos las **claves para obtener los mensajes**
- La expresión **`#{clave_mensaje}`**
 - Normalmente se asigna a texto **`th:text`** o enlace **`th:href`**

```
<div class="container" style="text-align: center">  
    <h2 th:text="#{welcome.message}"></h2>  
</div>
```



welcome.message=Bienvenidos a la página principal

Internacionalización > Cambio de idioma

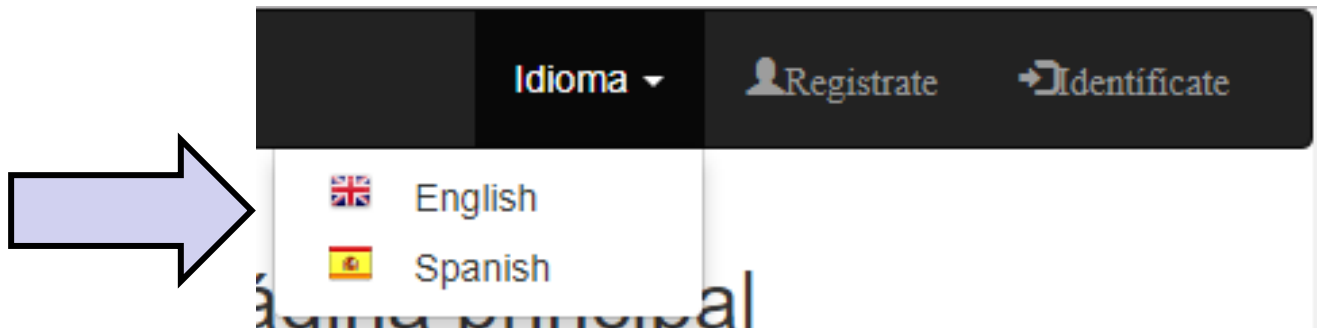
- Cambiar idioma
 - Se define un **parámetro** en la instancia del **LocaleChangeInterceptor**

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

- Parámetro definido en la configuración para el cambio de idioma
 - <http://miaplicacionweb.com/index?lang=en>
- El sistema selecciona los mensajes del fichero correspondiente al **idioma actual**
 - Ej. para ?lang=**en** se usa el fichero "message_**en**"
 - Para los códigos no definidos se usa el fichero por defecto "message"

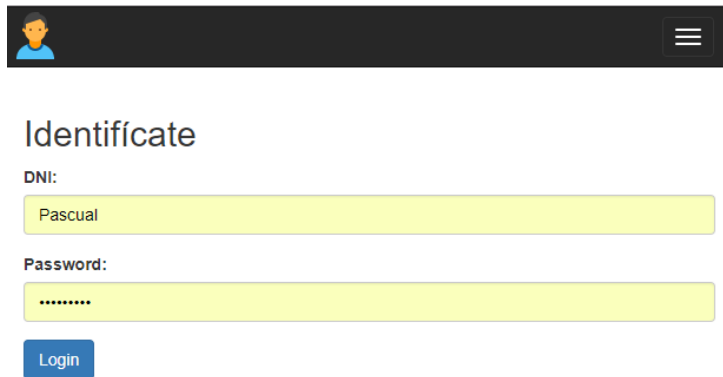
Internacionalización > Cambio de idioma

- Ejemplo de selección de idioma
 - Los botones envían una petición a la GET URL actual con el parámetro **lang=es** o **lang=en**



Spring Security > Autenticación

- **Autenticación:** proceso para validar la identidad de un usuario
- Los procesos más comunes verifican si existe coincidencia para un **identificador único de usuario** y una **contraseña**



A user interface for authentication. At the top, there is a dark header bar with a user icon on the left and a hamburger menu icon on the right. Below the header, the word "Identificate" is displayed. Underneath, there are two input fields: the first is labeled "DNI:" and contains the text "Pascual"; the second is labeled "Password:" and contains a series of dots. Below the password field is a blue button labeled "Login".



- El módulo **spring-boot-starter-security** incluye soporte para procesos de autenticación y autorización
 - Altamente configurable, desde muy simples a muy complejos
 - * La autenticación podría implementarse de otras formas

Spring Security > Autenticación

- Dependencia **spring-boot-starter-security**

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

- Definición de datos de autenticación

- **Identificador único**, "**username**" para Spring Security
 - Ej: el DNI, el email, etc.
- **Password** asociado al usuario, almacenado de forma segura (encriptación)
- ***Role** algunas aplicaciones definen permisos para diferentes tipos de usuarios
 - Ej: role profesor, role alumno, administrador, etc.

Spring Security > Encriptación

- El servicio que guarda los **usuarios** debe encriptar el password
- El objeto **BCryptPasswordEncoder** de Spring Security soporta la encriptación de forma ágil. hash Bcrypt:
<https://en.wikipedia.org/wiki/Bcrypt> (existen otros encoders)
 - Podemos usar **BCryptPasswordEncoder** como un Bean
Se instancia una vez y se inyecta donde se necesite

```
@Service
public class UserService {
    ...
    @Autowired
    private BCryptPasswordEncoder bCryptPasswordEncoder;

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

Spring Security > Encriptación

- El servicio inyecta **BCryptPasswordEncoder**
- ¿Dónde se instancia el Bean **BCryptPasswordEncoder** para poder ser inyectado?
 - No se trata de un componente (como los **@Controller**, **@Service**, etc.) que se instancia automáticamente en la anotación **@SpringBootApplication**
 - Debe ser instanciado como Bean **BCryptPasswordEncoder**
 - Se recomienda encapsular la configuración de seguridad en una clase

```
@Configuration
```

```
public class WebSecurityConfig extends ..... {
```

```
    @Bean
```

```
    public BCryptPasswordEncoder bCryptPasswordEncoder() {
```

```
        return new BCryptPasswordEncoder();
```

```
    }
```

Spring Security > Configuración

- Debemos especificar la configuración de Spring Security
- La clase debe ser hija de **WebSecurityConfigurerAdapter**
- Incluimos las anotaciones de clase:
 - **@Configuration**: clase de configuración
 - **@EnableWebSecurity**: activa Spring Web Security en la aplicación

```
@Configuration
@EnableWebSecurity

public class WebSecurityConfig extends
WebSecurityConfigurerAdapter {

    @Bean

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

Spring Security > Autorización (HttpSecurity)

- Sobrescribimos el método **configure(HttpSecurity http)**
 - Recibe un objeto **HttpSecurity** como parámetro
- **HttpSecurity** permite configurar el sistema de **autorización y otros aspectos de seguridad**
 - Definición de autorizaciones, URLs a las que pueden acceder:
 - Todos los usuarios
 - Usuarios autenticados
 - Usuarios autenticados con roles específicos
 - Formulario de autenticación / login
 - URL de formulario de autenticación
 - URL de redirección, página después de autenticación válida
 - URL de redirección, página después de autenticación inválida
 - Sistema “logout”, dejar de estar autenticado
 - URL de redirección, después de autenticación válida
 - Configuración de seguridad CSRF (habilitada por defecto)
 - Otros

Spring Security > Autorización (HttpSecurity)

- Definición de autorizaciones
 - **authorizeRequests()**: función principal/raíz
 - Dentro se anidan: (N) funciones **antMatchers** y (0-1) **anyRequest**
 - **antMatchers("URLS")**: especifica URL/s
 - Para cada **antMatchers** se especifica la autorización:
 - **permitAll()**: cualquier petición puede acceder
 - **authenticated()**: cualquier usuario autenticado puede acceder (independientemente del role)
 - **hasAuthority("Nombres de Roles")**: para acceder el usuario autenticado debe tener el **Role** especificado

Ej Implementación:

```
http
    .authorizeRequests ()
        .antMatchers ("/css/**", "/img/**", "/script/**").permitAll ()
        .antMatchers ("/principal", "/", "/registrarse").permitAll ()
        .antMatchers ("/notas/misnotas").authenticated ()
        .antMatchers ("/deletlesApp") .hasAuthority ("ROLE_ADMIN", "ROLE_PROFESOR")
        .antMatchers ("/usuarios/**").hasAuthority ("ROLE_ADMIN")
        .anyRequest ().authenticated ()
```

Spring Security > Autorización (HttpSecurity)

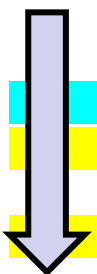
- Definición de autorizaciones

- Se basa en un **orden de prioridad**

Ej: La petición `/notas/misnotas` coincide con 3 matchers

- Sería gestionada por el primero `.antMatchers("/notas/misnotas")`
- Hace falta estar **autenticado** con **ROLE_PROFESOR**

http



```
.authorizeRequests()  
    .antMatchers("/css/**", "/img/**", "/script/**").permitAll()  
    .antMatchers("/principal", "/", "/registrarse").permitAll()  
    .antMatchers("/notas/misnotas").hasAuthority("ROLE_PROFESOR")  
    .antMatchers("/notas/**").authenticated()  
    .antMatchers("/usuarios/**").hasAuthority("ROLE_ADMIN")  
    .anyRequest().authenticated()
```

Spring Security > Autenticación (HttpSecurity)

- Formulario de autenticación / login
 - Función **formLogin()**, no se incluye en el **authorizeRequests()**
 - Los accesos no autorizados se direccionan automáticamente a **formLogin()**
 - Dentro se anidan:
 - **loginPage("URL")**: URL del formulario de login
 - Tipo de autenticación (**permitAll()**, **authenticated()** ...)
 - **defaultSuccessUrl ("URL")**: URL que se carga después de la autenticación valida
 - **failureUrl("URL")**: URL de carga en caso de fallo en la autenticación.
 - **failureHandler(authenticationFailureHandler())**: Manejador que captura el evento de fallo en la autenticación.

Ej Implementación:

http

```
.authorizeRequests()  
    .antMatchers("/css/**", "/img/**", "/script/**", "/", "/signup").permitAll()  
    .anyRequest().authenticated()  
    .and()  
    .formLogin()  
        .loginPage("/login")  
        .permitAll()  
        .defaultSuccessUrl("/principal")
```

Concatenador **and()**

Spring Security > Autorización (HttpSecurity)

- Sistema "logout", dejar de estar autenticado
 - Función **logout()**, habilita que los usuarios puedan cerrar sesión
 - Por defecto usa la URL **/logout**, (se puede modificar)
 - Dentro se anidan:
 - Tipo de autenticación (**permitAll()**, **authenticated()** ...)
 - **logoutSucessUrl("URL")**: redirección después de cerrar sesión
 - Otros

Ej Implementación:

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

Concatenador **and()**



Spring Security > CSRF

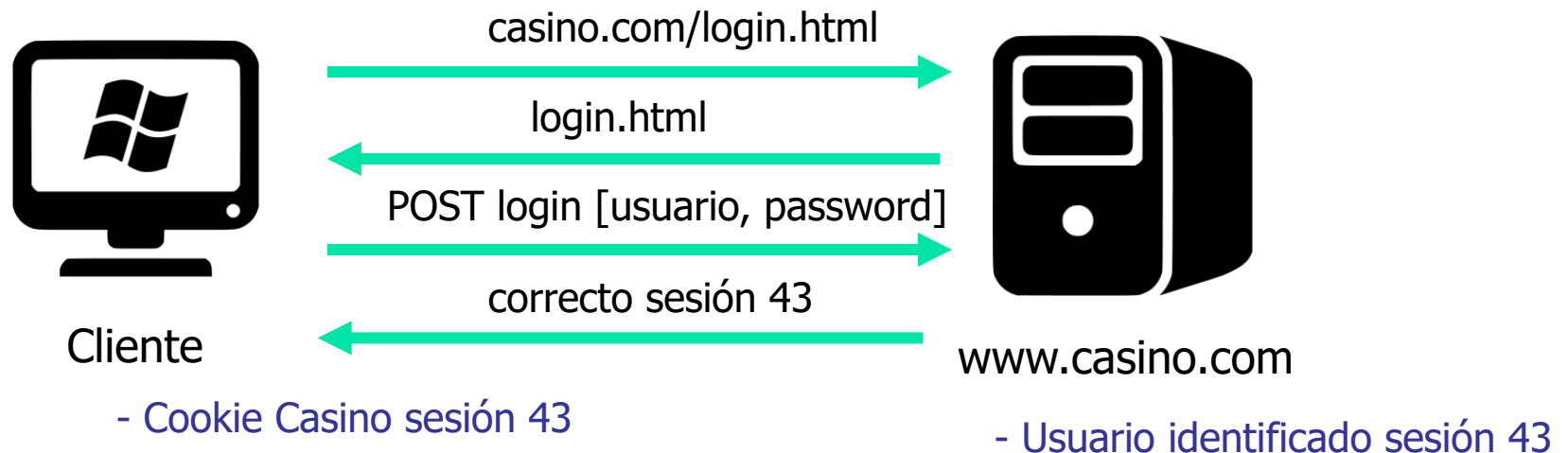
- Configuración de seguridad **CSRF** por defecto activada
 - Función **csrf()** habilita la configuración de seguridad csrf
 - Para usar esta seguridad por defecto no hace falta incluir nada
 - * Se podría configurar de formas más específicas
 - Para desactivarla se utiliza la función **disable()**

Ej Implementación:

```
.logout ()  
    .permitAll ()  
    .logoutSucessUrl ("/despedida")  
    .and ()  
.csrf () .disable () ;
```

Spring Security > CSRF

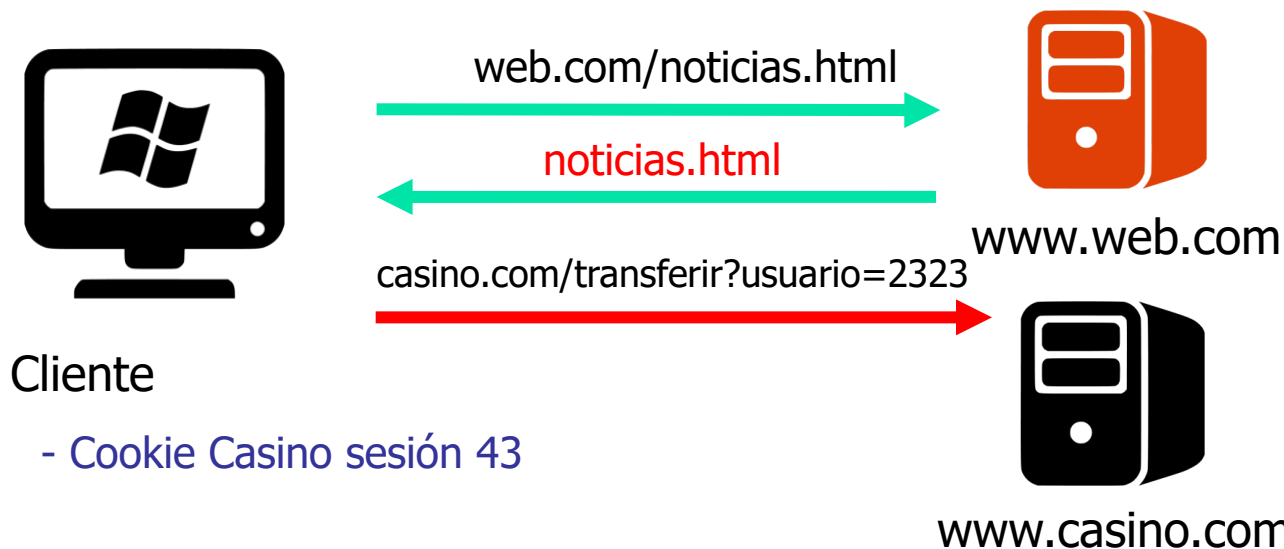
- Falsificación de petición en sitios cruzados **CSRF** (Cross-site request forgery)
- Exploit basado en que un cliente envía peticiones no intencionadas y de carácter malicioso a un sitio web en el que confía
- Ejemplo
 - Nos autenticamos en www.casino.com
 - El cliente recibe una cookie de sesión y el servidor abre una sesión para el usuario



Spring Security > CSRF



■ Ejemplo

- El cliente continua navegando en varias páginas
 - Por ejemplo, un link compartido en un comentario www.web.com/noticias.html
- La página **noticias.html** se carga en el navegador del usuario
 - Contiene código JavaScript con una petición www.casino.com/transferir?usuario=2323
 - Seguramente una petición en background no perceptible
- El **cliente** ejecuto realizo sin saberlo una petición no deseada
 - Al estar autenticado en www.casino.com la petición fue aceptada



Spring Security > CSRF

- Los **tokens CSRF** son una medida de protección efectiva
- Las paginas sensibles que el **servidor** envía al **cliente** contienen un **token CSRF**.
 - **Token** generado y gestionado por el framework
 - Normalmente enviado como un campo oculto de un formulario
- El **cliente** envía ese mismo **token CSRF** en su siguiente petición
 - Sí no hay token o no concuerda, **se rechaza la petición**
- Se verifica que el cliente envió la petición desde el propio sitio



Identifícate

DNI:

Password:

```
▼<form action="/login" class="form-horizontal" method="post">  
  ▶<div class="form-group">...</div>  
  ▶<div class="form-group">...</div>  
  ▶<div class="form-group">...</div>  
  <input type="hidden" name="_csrf" value="e2299792-2c2c-4eb2-9705-6396ad4ff0bc">  
</form>
```

Spring Security > Tokens CSRF

- Se debe incluir tokens CSRF en:
 - Login, logout, todos los formularios (otras acciones sensibles)
- Para habilitarlos se debe:
 1. Eliminar **.csrf().disable()** de la configuración de seguridad

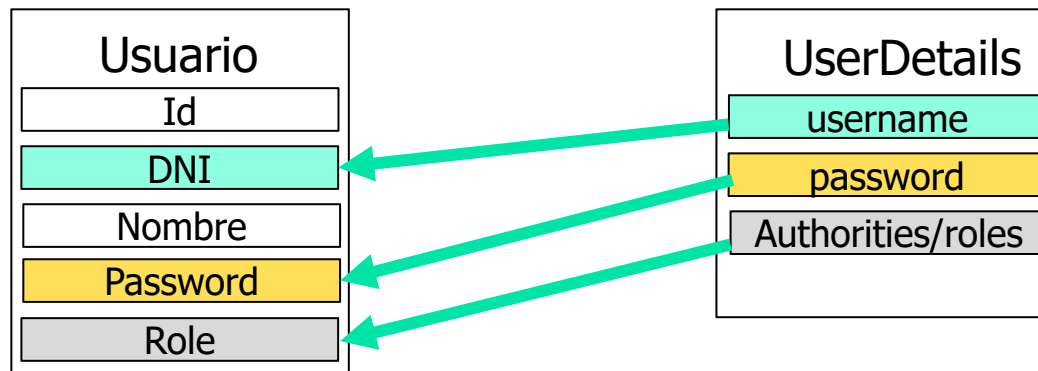
```
.logout ()  
    .permitAll ()  
    .logoutSucessUrl ("/despedida")  
    .and ()  
    .csrf ().disable ();
```

2. Agregar el token como campo oculto (login, logout, formularios, otros).
 - La variable **_csrf** contiene los dos atributos necesarios **parameterName** y **token**
 - Accesible directamente desde las vistas (no hay que modificar los controladores)

```
<button type="submit" class="btn btn-primary">Login</button>  
<input type="hidden" th:name="${_csrf.parameterName}" th:value="${_csrf.token}"/>  
</form>
```

Spring Security > UserDetails

- La información del usuario relativa a la autenticación se almacena en la base de datos de forma estándar
 - Un **identificador único** del usuario (Nickname, DNI, mail, UO)
 - Depende de la lógica de la aplicación
 - Un **password**, debe guardarse encriptado
 - Por ejemplo usando el Bean **BCryptPasswordEncoder**
 - **Role/s** del usuario
 - Algunas aplicaciones no guardan **role/s**, todos tienen el mismo
- Spring Security no procesa usuarios de la lógica, sino objetos **UserDetails** contiene únicamente la información de autenticación




Spring Security > UserDetailsService

- **UserDetailsService** será el encargado de crear los **UserDetails**
- Sobrescribe la función **loadUserByUsername(username)**
- **loadUserByUsername** debe:
 1. Obtener el **usuario** asociado al **username** (en el repositorio)
 - Con la referencia al usuario se comparará el **password**
 2. Crear una colección de tipo **GrantedAuthority**
 - En esta colección se almacenan los roles
 - Ej1: todos los usuarios tienen el mismo rol
 - Ej2: cada usuario tiene un rol específico (nombre del rol en la base de datos)
 - Ej3: cada usuario tiene varios roles (nombres de los roles en la base de datos)
 3. Crea un objeto de tipo **UserDetails** (Spring Security)
 - Contiene el **username**, **password** y **colección de roles**
 4. Retorna el objeto **UserDetails**

Spring Security > UserDetailsService

■ Implementación

Elegimos un parámetro único como "username"



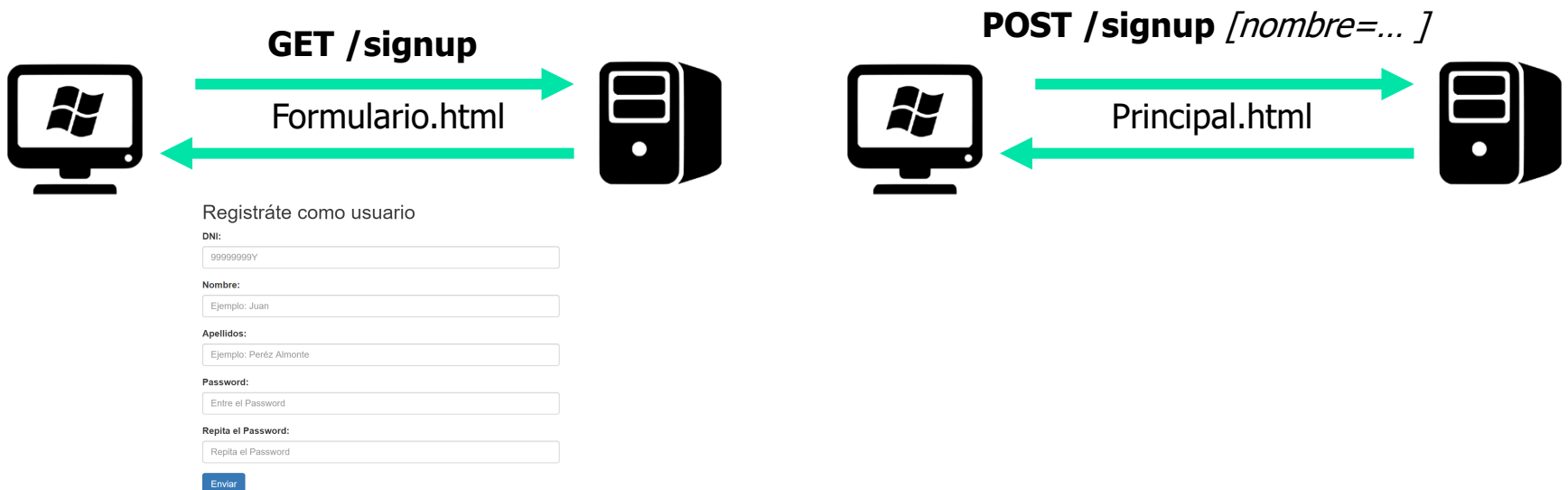
```
@Override
public UserDetails loadUserByUsername(String dni) throws UsernameNotFoundException{
    // 1 Usuario en la aplicación
    User user = usersRepository.findByDni(dni);

    // 2 Configurar Autoridades / EJ,TODOS EL ROLE_ESTUDIANTE
    Set<GrantedAuthority> grantedAuthorities = new HashSet<>();
    grantedAuthorities.add(new SimpleGrantedAuthority("ROLE_ESTUDIANTE"));

    // 3 Construir y retornar el "Userdetails"
    return new org.springframework.security.core.userdetails.User(
        user.getDni(), user.getPassword(), grantedAuthorities);
}
```

Spring Security > Controladores

- Los **controladores** procesan peticiones relativas a la autenticación
- Un esquema común en Spring Security podría ser el siguiente:
 - **GET /signup:** mostrar la vista que solicita los datos de registro de un usuario
 - **POST /signup:** registrar un usuario en la aplicación
 - **GET /login:** mostrar la vista que solicita los datos de autenticación



Spring Security > Controladores

- ¿Se implementa en el controlador una respuesta a POST login?
 - **POST /login:** intentar autenticar el usuario en la aplicación
 - **GET /logout:** dejar de estar autenticado
 - **No se implementa**, esta definido en Spring Security
- Spring Security necesita usar un formulario de autenticación el cual debe de:
 - Ser enviado con el método **POST** a la URL **/login**
 - Contener un **input** con nombre **username**
 - Contener un **input** con nombre **password**
 - *Sí habilitamos la seguridad csrf debe incluirse el campo **_csrf**
 - Los nombres de los parámetros **username** y **password** son obligatorios, coinciden con los parámetros del objeto **UserDetails**

Spring Security > Lógica de negocio

- El **SecurityContextHolder** da acceso al usuario autenticado desde toda la aplicación
- La lógica de negocio suele necesitar identificar al usuario autenticado
 - **getContext().getAuthentication()** retorna el objeto **Authentication**
 - **Authentication** contiene información sobre la autenticación
 - **.getName()** username del usuario autenticado
 - **.getAuthorities()** autoridades/roles del usuario autenticado
 - **.setAuthenticated(Boolean)** desautentica al usuario manualmente
 - Otros
- Ejemplo

```
Authentication auth = SecurityContextHolder.getContext().getAuthentication();  
String dni = auth.getName();
```

Autenticación > Thymeleaf

- Módulo de extensión de **thymeleaf-extras-springsecurity4**

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

- Acceso a los elementos de Spring Security desde las plantillas
 - Simplifica algunas labores de desarrollo
 - Evita el envío de datos de seguridad del controlador a la plantilla
- Atributo **sec:authentication** información sobre el cliente autenticado
 - **sec:authentication="principal.username"** consultar el username del cliente autenticado
- Atributo **sec:authorize** información sobre autorizaciones
Se utiliza para mostrar HTML condicional al nivel de autorización
 - **sec:authorize="isAuthenticated()"** valida si el cliente esta autenticado
 - **sec:authorize="hasRole('NombreRole')"** valida si el cliente esta autenticado

Autenticación > Thymeleaf

- Implementación **sec:authentication**
 - Muestra el valor del **principal.rolename** en un span

```
<p>  
  Usuario <span sec:authentication="principal.username"></span>  
</p>
```

- Implementación **sec:authorize**
 - El bloque se muestra si el cliente no está autenticado

```
<li sec:authorize="!isAuthenticated()">  
  <a href="/login" th:text="#{login.message}">  
    Identifícate  
  </a>  
</li>
```

Spring Security > Fuerza bruta

- Probar una gran cantidad de **contraseñas** para un usuario
 - Obtenidas en diccionarios de datos
 - Generadas secuencialmente
- **Detectar** cuando un usuario intenta identificarse repetidamente sin éxito
 - Acción de seguridad: notificárselo al usuario, banear la IP, suspender el usuario temporalmente, etc.
- La **detección** de fuerza bruta se puede implementar:
 - De forma más o menos manual
 - Usando objetos de Spring Security
 - Implementando componentes que escuchen **AuthenticationFailureBadCredentialsEvent** y **AuthenticationSuccessEvent** que registran los intentos de autenticación

Spring Security > Fuerza bruta

- Implementación, componente que escucha eventos **AuthenticationFailureBadCredentialsEvent**
 - Sobrescribir **onApplicationEvent**
 - Consultar la información del usuario y el acceso

```
@Component
public class AuthenticationFailureListener implements
ApplicationListener<AuthenticationFailureBadCredentialsEvent> {

    @Override
    public void onApplicationEvent(AuthenticationFailureBadCredentialsEvent e) {
        String username = e.getAuthentication().getName();

        WebAuthenticationDetails detalles =
            (WebAuthenticationDetails) e.getAuthentication().getDetails();
        String ip = detalles.getRemoteAddress();
        String idSession = detalles.getSessionId();

    }
}
```


Repositorios > JPA Entidades relacionadas

- Mapeo de **Entidades** usando las anotaciones de **java.persistence**
- Ejemplo de relación entre entidades
 - Uno a muchos / Muchos a uno, **@OneToMany** y **@ManyToOne**
 - Ej. Un usuario muchas notas

```
@Entity
public class User {
    @Id
    @GeneratedValue
    private long id;
    ...

    @OneToMany(mappedBy = "user",
                cascade = CascadeType.ALL)
    private Set<Mark> marks;
```

1

```
@Entity
public class Mark {
    @Id
    @GeneratedValue
    private Long id;
    ...

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

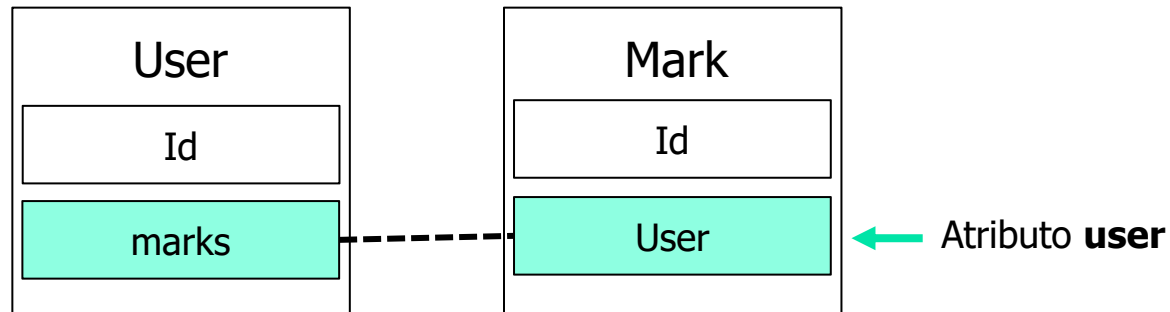
N

Repositorios > JPA Entidades relacionadas

■ Entidad 1 User @OneToMany

- Se almacenan en una colección
- Atributo **mappedBy = user** indica que esta relación fue construida añadiendo un atributo **user** en la clase **Mark**

```
@OneToMany (mappedBy = "user", cascade = CascadeType.ALL)  
private Set<Mark> marks;
```



- Atributo **cascade = CascadeType.ALL**, cuando la entidad es guardada, eliminada o actualizada, su entidad relacionada también lo es
 - Otros valores más restrictivos: `CascadeType.PERSIST`, `CascadeType.REMOVE`, etc.

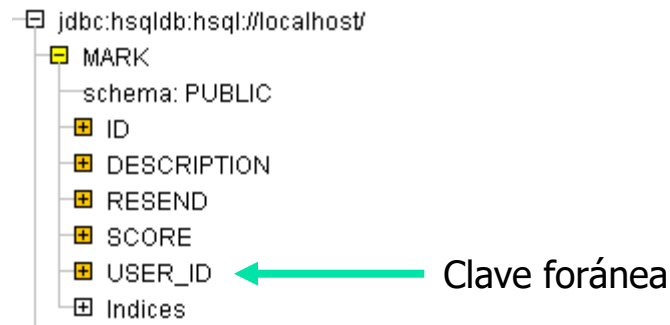
Repositorios > JPA Entidades relacionadas

■ Entidad 2 Mark @ManyToMany

- Se almacenan en un objeto
- Anotación **@JoinColumn**, especifica que esta columna es una asociación de entidades
 - Podemos especificar un nombre con el atributo **name**

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

■ Esquema de datos



Repositorios > JPA Entidades relacionadas

- Ejemplo de relación entre entidades
 - Muchos a muchos **@ManyToMany**
 - Ej. Un libro con muchos escritores, un escritor con muchos libros

```
@Entity
public class Writer{
    @Id
    @GeneratedValue
    private Long id;
    ...

    @ManyToMany(mappedBy = "publishers")
    public Set<Book> books;
```

N

```
@Entity
public class Book{
    @Id
    @GeneratedValue
    private long id;
    ...

    @ManyToMany(cascade = CascadeType.ALL)
    @JoinTable(name = "books_writers",
        joinColumns = { @JoinColumn(name = "book_id") },
        inverseJoinColumns = { @JoinColumn(name = "writer_id") }
    )
    public Set<Publisher> publishers;
}
```

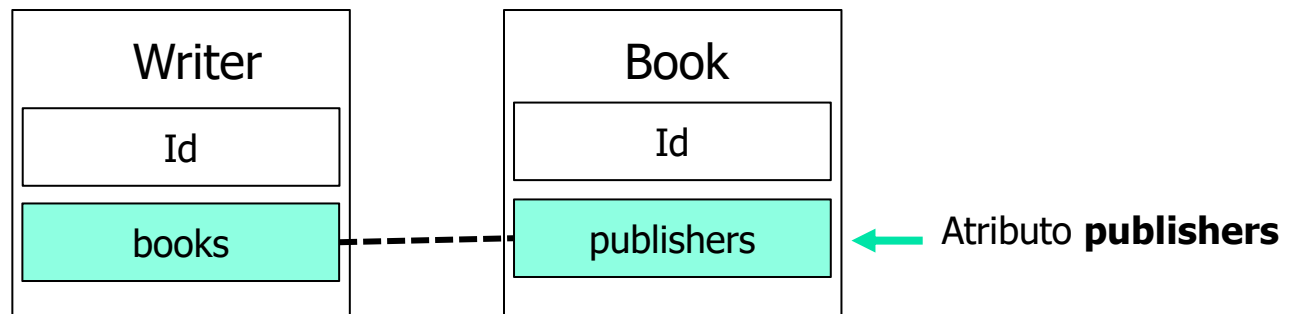
N

Repositorios > JPA Entidades relacionadas

■ Entidad 1 Writer @ManyToMany

- Se almacenan en una colección
- Atributo **mappedBy = publishers** indica que esta relación fue construida añadiendo un atributo **publishers** en la clase **Book**

```
@ManyToMany(mappedBy = "publishers")  
public Set<Book> books;
```



Repositorios > JPA Entidades relacionadas

■ Entidad 2 Book @ManyToMany

- Se almacenan en una colección
- Anotación **@JoinTable**, especifica la estructura de la tabla que almacena la relación entre ambas entidades

```
@ManyToMany(cascade = CascadeType.ALL)
@JoinTable(name = "books_writers",
    joinColumns = { @JoinColumn(name = "book_id") },
    inverseJoinColumns = { @JoinColumn(name = "writer_id") }
)
public Set<Publisher> publishers;
```

- Esquema de datos



Repositorios > Datos de prueba

- Definición de un servicio para insertar datos
 - Creamos un servicio **InsertSampleDataService**
 - Inyectamos los **repositorios** en los que queramos agregar datos
 - Podría ser uno o varios

```
@Autowired  
private UserService userService;
```

- Definimos una función **@PostConstruct** para agregar datos
 - Se ejecuta después de instanciar los servicios

```
@PostConstruct  
public void init() {  
    User user1 = new User("99999990A", "Pedro", "Díaz");  
    userService.addUser(user1);  
}
```

Repositorios > Datos de prueba

- Inicialización de la base de datos (JPA)
 - Propiedades **application.properties**
 - Reiniciar el contenido de la base de datos en cada ejecución
 - *En alguna ocasión es conveniente borrarla manualmente para que funcione bien

```
spring.jpa.hibernate.ddl-auto=create
```
 - Validar y mantener los datos que se encuentran en la base de datos

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


Validación de datos > Introducción

- Los datos introducidos por los usuarios deben ser **validados**
- Consiste en comprobar que los datos son adecuados
 - Formato, longitud, cumplimiento de reglas de lógica de negocio, etc.
 - Se suele **informar al usuario** cuando introduce datos incorrectos
- Es positiva para la **experiencia de usuario** y la **seguridad** del sistema
 - Notifica cuando se introducen datos no validos
 - Protege contra datos maliciosos que podrían dañar el sistema

Regístrate como usuario

DNI:	<input type="text" value="99999999Y"/>
<small>Este campo no puede ser vacío El DNI debe tener entre 5 y 24 caracteres.</small>	
Nombre:	<input type="text" value="Ejemplo: Juan"/>
<small>El nombre debe tener entre 5 y 24 caracteres.</small>	
Apellidos:	<input type="text" value="Ejemplo: Pérez Almonte"/>
<small>El apellido debe tener entre 5 y 24 caracteres.</small>	
Password:	<input type="password" value="Entre el Password"/>
<small>La contraseña debe tener entre 5 y 24 caracteres.</small>	
Repita el Password:	<input type="password" value="Repita el Password"/>
<input type="button" value="Enviar"/>	

Validación de datos > Introducción

- Tipos de validaciones (no excluyentes entre ellos)
 - **Validaciones en el cliente:** utilizan código de script que se ejecuta en el navegador
 - Validan los datos **antes de enviarlos al servidor**
 - Evitan enviar al servidor una petición claramente invalida
 - En muchos casos se centran en formato y longitud
 - Muy dinámicas y **positivas para la experiencia de usuario**
 - Pueden ser fácilmente desactivadas (seguridad)
 - **Validaciones en el servidor:** comprobaciones realizadas en el servidor sobre los datos enviados por el usuario
 - Validan los datos **al recibir la petición**
 - Pueden validar **condiciones de lógica de negocio**
 - EJ, Mail no replicado en el sistema, ID asociada al producto existe realmente
 - Como resultado de la validación se podrían generar diferentes respuestas
 - No pueden ser desactivadas por el cliente

Validación de datos > Validador

- La interfaz **Validator** permite construir validadores en Spring
- Los **validators** añaden validación en el **servidor**
 - Dispone de un sistema para mostrar los mensajes de error en la vista
- Se recomienda implementar un **validator** por proceso/formulario
- Implementación de una clase **validator**
 - Debe implementar la interfaz **validator**
 - Se gestionan como **Bean**, algunas opciones para obtenerlo:
 - Usar **@Component** genérico (sin estereotipo)
 - Usar una clase Java estándar e instanciarla con **@Bean**
 - Sobrescribir la función **validate(Object, Errors)**
 - **Object** contiene el objeto creado con los campos del formulario
 - * *Se suele corresponder con una entidad, Ej: usuario, nota, etc.*
 - **Errors** para almacenar los mensajes de error (y a que campo van asociados)

Validación de datos > Validador

- La función **validate(Object, Errors)**
 - **objeto** contiene el objeto creado con los campos del formulario
 - El controlador que recibía el formulario usaba **@ModelAttribute <Entidad>**
Los parámetros se guardan en una entidad (coincidencia por nombre)

```
@RequestMapping(value = "/signup", method = RequestMethod.POST)
public String setUser(@ModelAttribute User user, Model model) {
```

- Validación de los parámetros
 - **Correcto** -> no requiere acción
 - **Incorrecto** -> almacenar el error en la variable **errors**
 - **errors.rejectValue(<clave_del_campo> , <mensaje_de_error>)**

```
@Override
public void validate(Object target, Errors errors) {
    User user = (User) target;
    if (user.getDni().length() < 5 || user.getDni().length() > 24)
        errors.rejectValue("dni", "El DNI debe tener entre 5 y 24 caracteres");
    if (userService.getUserByDni(user.getDni()) != null) {
        errors.rejectValue("dni", "El DNI ya esta siendo usado");
    }
}
```

Validación de datos > Mostrar errores

- Varias opciones para **mostrar errores en la vista**
- Opción para modificar la plantilla:
 - Incluir un bloque HTML condicional junto a cada campo
 - **th:if="expresión"** muestra condicionalmente bloques HTML
 - **#field.hasErrors('<clave_del_campo>')**, objeto de utilidades, retorna si hay errores asociados al campo (true/false)
 - **th:errors="*{<clave_del_campo>}"** muestra la lista de mensajes de error asociados a un atributo de un objeto
 - Un mismo campo podría tener N mensajes de error asociados

validator

```
errors.rejectValue("dni",  
    "El DNI debe tener entre 5 y 24 caracteres");  
errors.rejectValue("dni",  
    "El DNI no puede contener dos letras");
```

plantilla

```
<input type="text" class="form-control"  
    name="dni" placeholder="99999999Y" />  
<span th:if="${#fields.hasErrors('dni')}"  
    th:errors="*{dni}">
```

DNI:

99999999Y

El DNI debe tener entre 5 y 24 caracteres.

El DNI no puede contener dos letras.

Validación de datos > Agregar el validador

- Agregar el **validator**. Carga del formulario
 - El **controlador** que responde el formulario debe:
 - Incluir una **entidad vacía** en el **modelo** enviado a la vista

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

- La **plantilla** que contiene el formulario debe:
 - Incluir **entidad vacía** enviada por el controlador en el formulario
 - **th:object** atributo del modelo sobre el que se pueden ejecutar comandos, necesario para usar los comandos **th:errors** (están asociados a un objeto)

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

Validación de datos > Agregar el validador

- Usar el **validator** al recibir los datos del formulario
 - Vinculación de una **petición** con un **validator**
 - Inyectamos el Bean **validador** en un **controlador**

```
@Autowired
private SignUpFormValidator signUpFormValidator;
```

- El controlador debe recibir información adicional de validación
 - **@Validated** indica el atributo del modelo a validar
 - **BindingResults** contiene los resultados de la validación
 - Define el método **hasErrors()**, indica si se produjo algún error (true/false)

```
@RequestMapping(value = "/signup", method = RequestMethod.POST)
public String signup(@ModelAttribute @Validated User user, BindingResult result ...
    signUpFormValidator.validate(user, result);
    if (result.hasErrors()) {
        return "signup";
    }
    user.setRole(rolesService.getRoles()[0]);
    userService.addUser(user);
    securityService.autoLogin(user.getDni(), user.getPasswordConfirm());
    return "redirect:home"; //Se accede la vista de operaciones autorizadas/privadas
}
```

Validación de datos > ValidationUtils

- **ValidationUtils** clase estática de utilidades de validación
 - Varios métodos para realizar validaciones comunes
 - **rejectIfEmpty(errors, <clave_del_campo>', mensaje error)** incluye el mensaje de error si el campo es vacío
 - **rejectIfEmptyOrWhiteSpace (errors, <clave_del_campo>', mensaje error)** incluye el mensaje de error si el campo es vacío o solo contiene espacios en blanco

@Override

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