

Spring

Spring Security

Spring: Spring Security

- Lo primero que debemos saber es que Spring Security es un framework aparte de Spring.
- Segundo como es un proyecto aparte de Spring Framework, debemos agregarlos como dependencia a nuestro POM.

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

Spring: Spring Security

- Ahora conversemos de seguridad.
- Que es lo mas básico que debemos proteger:
 - Acceso autenticado a nuestra aplicación.
 - ¿Quienes pueden utilizar mi aplicación?
 - Acceso autorizados a nuestros recursos.
 - ¿Que pueden utilizar de mi aplicación?

```
<dependency>
```

```
<groupId>org.springframework.boot  
</groupId>
```

```
<artifactId>spring-boot-starter-  
security</artifactId>
```

```
</dependency>
```

Spring: Spring Security

- Autenticación

- La versión más simple de este proceso, es el login.
- También podríamos agregar CORS (Cross-Origin Request Sharing). Para evitar el csrf (cross-site request forgery)
- O incluso un simple token de acceso.

```
<dependency>
```

```
<groupId>org.springframework.bo  
ot</groupId>
```

```
<artifactId>spring-boot-  
starter-security</artifactId>
```

```
</dependency>
```

Spring: Spring Security

- Autorización

- Para esto tenemos algunas opciones para implementar como RBAC, pero Spring Security provee ACL (Access Control List)
- Simplemente una división lógica de los tipos de usuarios de nuestra aplicación
- Y una lista con los accesos a recursos autorizados, para cada uno de los tipos de usuario.

```
<dependency>
```

```
<groupId>org.springframework.bo  
ot</groupId>
```

```
<artifactId>spring-boot-  
starter-security</artifactId>
```

```
</dependency>
```

Spring: Spring Security

- Ahora bien. Spring Security, trabaja con una estructura por defecto tanto de registro de los datos autenticación, autorización, como de las acciones a realizar en cada caso.
- Para las estructuras de datos, trabaja con la tabla Users, y Authorities, con sus respectivas columnas.

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

Spring: Spring Security

- Ahora esto lo debemos integrar con Thymeleaf.
- La dependencia a utilizar va a depender de la versión de Spring que estemos ejecutando.
- Para la versión que estamos utilizando utilizaremos la siguiente dependencia.

```
<dependency>  
<groupId> org.thymeleaf.extras  
</groupId>  
<artifactId> thymeleaf-extras-  
springsecurity5 </artifactId>  
</dependency>
```

Spring: Validation

- Ahora que hemos agregado nuestras dependencias. Si ejecutamos nuestra aplicación, lo primero que nos solicitará es autenticarnos.
- Pero no hemos implementado nada de eso. Pues manos al código.

Please sign in

Sign in

Spring: Validation

- Paso 1: Lo primero es crear la clase SpringSecurityConfig. No debe ser exactamente este nombre pero es casi un estandar. Ahora por buenas practicas de Spring debemos poner el postfijo Config.

```
@Configuration  
public class  
SpringSecurityConfig extends  
WebSecurityConfigurerAdapter  
{  
  
}
```

Spring: Validation

- Paso 2: Como vamos a utilizar JPA para acceder a los usuarios registrados en la base de datos. Debemos agregar el servicio de acceso a los usuarios.

@Configuration

```
public class  
SpringSecurityConfig extends  
WebSecurityConfigurerAdapter {
```

@Autowired

```
private  
UserDetailsServiceImplementation  
userService;
```

@Bean

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

Spring: Validation

- Paso 3: Definimos el tipo de encriptación de las claves de nuestros usuarios.

@Configuration

```
public class SpringSecurityConfig extends  
WebSecurityConfigurerAdapter {
```

```
...
```

@Autowired

```
public void  
configurerGlobal(AuthenticationManagerBuild  
er build) throws Exception  
{  
    build.userDetailsService(userDetailsService  
)  
    .passwordEncoder(this.passwordEncoder());  
}
```

Spring: Validation

- Paso 4: Definimos el servicio que permitirá a SpringSecurity tener acceso a nuestros usuarios y verificar sus credenciales.
- El servicio deberá implementar UserDetailsService.
- Esto para sobre escribir el método loadUserByUsername, y así manejar nosotros la búsqueda del usuario en nuestras tablas.

```
@Service("UserDetailsService")
```

```
public class UserDetailsServiceImpl  
    implements UserDetailsService{  
    ...  
}
```

Spring: Validation

- Paso 5: Sobre escribiendo el método loadUserByUsername.
- Paso 5.1: Primero con acceso a nuestro repositorio de usuario JPA buscamos un usuario con el username que nos proveen como parámetro
- Paso 5.2: Segundo si no lo encontramos lanzamos una exception informando que el usuario no existe.
- Paso 5.3: Si existe entonces todo sigue normal y debemos buscar sus Authorities en el sistema.

```
@Override  
  
@Transactional(readOnly=true)  
  
public UserDetails loadUserByUsername(String  
username) throws UsernameNotFoundException {  
  
    cl.tswoo.ecommerce.models.User user =  
    userRepository.findByUsername(username);  
  
    if(user == null) {  
  
        throw new  
UsernameNotFoundException("Username: " +  
username + " no existe en el sistema!");  
    }  
}
```

Spring: Validation

- Paso 5.4: Buscando los Authorities del usuario encontrado.
- Paso 5.5: Si no tiene ninguna Authority, se lanza una Excepción.

```
@Override
@Transactional(readOnly=true)
public UserDetails loadUserByUsername(String username) throws
UsernameNotFoundException {

    cl.tswoo.ecommerce.models.User user =
userRepository.findByUsername(username);

    if(user == null) {

        throw new UsernameNotFoundException("Username: " +
username + " no existe en el sistema!");

    }

    List<GrantedAuthority> authorities = new
ArrayList<GrantedAuthority>()

    for(Role role: user.getRoles()) {

        authorities.add(new
SimpleGrantedAuthority(role.getAuthority()));

    }

    if(authorities.isEmpty()) {

        throw new UsernameNotFoundException("Error en el Login:
usuario '" + username + "' no tiene roles asignados!");

    }

}
```

Spring: Validation

- Paso 5.6:Retorna un User de SpringSecurity con el username, password, si esta habilitado, y sus Authorities correspondientes.

```
@Override
@Transactional(readOnly=true)
public UserDetails loadUserByUsername(String username) throws
UsernameNotFoundException {
    cl.tswoo.ecommerce.models.User user =
    userRepository.findByUsername(username);

    if(user == null) {
        throw new UsernameNotFoundException("Username: " + username +
        " no existe en el sistema!");
    }

    List<GrantedAuthority> authorities = new
    ArrayList<GrantedAuthority>()

    for(Role role: user.getRoles()) {
        authorities.add(new
        SimpleGrantedAuthority(role.getAuthority()));
    }

    if(authorities.isEmpty()) {
        throw new UsernameNotFoundException("Error en el Login:
        usuario " + username + " no tiene roles asignados!");
    }

    return new
    org.springframework.security.core.userdetails.User(user.getUsername(),
    user.getPassword(), user.getEnabled(), true, true, true, authorities);
}
```

Spring: Validation

- Paso 6: Configurar el acceso a los recursos.
- Para esto haremos un Override del método configure.
- Permitiremos el acceso a todos los recursos estáticos.
- Permitiremos a todos el acceso al login.
- Permitiremos a todos el acceso al logout.
- Definiremos una pagina custom para informar de un acceso denegado a un recurso.

```
@Override
```

```
protected void configure(HttpSecurity http)  
throws Exception {
```

```
http.authorizeRequests()
```

```
.antMatchers("/", "/css/**", "/js/**", "/images/  
**").permitAll()
```

```
.anyRequest().authenticated()
```

```
.and()
```

```
.formLogin().permitAll()
```

```
.and()
```

```
.logout().permitAll()
```

```
.and()
```

```
.exceptionHandling()
```

```
.accessDeniedPage("/access-denied");
```

```
}
```


Spring: Validation

- Paso 7: Para que se pueda mostrar la pagina custom de acceso denegado debemos agregar en el MvcConfig, un controllerViewRegistry especifico para la pagina custom.
- Adicionalmente agregar la pagina a los templates del proyecto.
- Es un método para ahorrarnos el construir un controller que resuelva la llamada al recurso.

@Configuration

```
public class MvcConfig implements  
WebMvcConfigurer {
```

```
...
```

```
public void  
addViewControllers(ViewControllerRegistry  
registry) {
```

```
    registry.addViewController("/access-  
denied").setViewName("access-denied");
```

```
}
```

```
}
```

Spring: Validation

- Listo!
- Spring Security se encarga de validar el usuario y sus Authorities.
- Ahora nunca especificamos a que va a tener acceso cada Authority.

Spring: Verification

- Ahora comenzamos a especificar a que va a tener acceso cada Authority.
- Esta especificación la podemos hacer en dos lados: View y/o Controller

Spring: Verification in Controller

- Podemos especificar el acceso a cada recurso, o especificar en general para el controller.
- La anotación `@Secured` de SpringSecurity permite especificar que rol o roles tienen acceso al o los recursos.

```
@RequestMapping("/customer")
@Controller
public class CustomerController {

    @Secured("ROLE_ADMIN")
    @GetMapping("/create")
    public String create(Model model) {
        Customer customer = new Customer();
        model.addAttribute("customer", customer);
        return "customer/form";
    }

    @Secured({"ROLE_USER", "ROLE_ADMIN"})
    @GetMapping("/detail")
    public String detail(@RequestParam Integer id, Model model) {
        Customer customer = service.getById(id);
        model.addAttribute("customer", customer);
        return "customer/detail";
    }
}
```

Spring: Verification in Controller

- Podemos especificar el acceso a cada recurso, o especificar en general para el controller.
- La ausencia de la especificación de la anotación `@Secured` permite que todos tengan acceso al recurso.

```
@RequestMapping("/customer")  
  
@Controller  
  
public class CustomerController {  
  
    @GetMapping({"/", "/list"})  
  
    public String list(Model model) {  
  
        List<Customer> list = service.listAll();  
  
        model.addAttribute("customers", list);  
  
        return "customer/list";  
  
    }  
  
}
```

Spring: Verification in View

- Para restringir el acceso en la vista a elementos de esta, debemos usar los elementos del namespace que nos provee thymeleaf de springSecurity.
- Pero antes debemos agregar el namespace.

```
<html xmlns:th="http://www.thymeleaf.org"
xmlns:sec="http://www.thymeleaf.org/extras/spring-security">
```

Spring: Verification in View

- Ahora para restringir el acceso a elementos de la vista debemos agregar, el modificador de comportamiento
sec:authorize="hasRole('ROLE_USER')"
- or si son varios rol que tienen acceso al elemento de la vista
sec:authorize="hasAnyRole('ROLE_USER','ROLE_ADMIN')"

```
<a sec:authorize="hasRole('ROLE_ADMIN')"  
href="/customer/create" class="btn btn-primary"  
th:text="#{text.customer.list.new}"></a>
```