

# Curso de Spring Framework

## Tema 4 Seguridad con Spring

**Micael Gallego**

micael.gallego@gmail.com

@micael\_gallego

- **Introducción**
- **Https**
- **Spring Security**
- **API REST para web SPA con seguridad**

- **Servicios de seguridad**
  - Un **servicio de seguridad** protege las comunicaciones de los usuarios ante determinados ataques. Los principales son:
    - **Autenticación (authentication):** sirve para garantizar que una entidad (persona o máquina) es quien dice ser
    - **Autorización (authorization):** sirve para discernir si una entidad tiene acceso a un recurso determinado

- **Servicios de seguridad**
  - Un **servicio de seguridad** protege las comunicaciones de los usuarios ante determinados ataques. Los principales son:
    - **Integridad (data integrity):** garantiza al receptor del mensaje que los datos recibidos coinciden exactamente con los enviados por el emisor
    - **Confidencialidad (data confidentiality)** proporciona protección para evitar que los datos sean revelados a un usuario no autorizado

- **Autenticación**

- La autenticación se consigue mediante:
  - **Algo que sabes.** Por ejemplo, unas credenciales login-password
  - **Algo que tienes.** Por ejemplo, una tarjeta de acceso
  - **Algo que eres.** Por ejemplo, cualidades biométricas (huella digital...)

- **Autorización**

- La autorización determina si un **usuario puede acceder** a un recurso determinado en base a permisos (*grants*), lista de control de acceso (*Access Control List, ACL*), políticas (*policies*), roles, tokens, ...
- A veces requiere **autenticación** previa (es decir, confirmar la **identidad** del usuario)

- **Integridad**

- La integridad se consigue típicamente con funciones **Hash (resumen)**
- Son funciones que convierten un **texto plano** en una **secuencia alfanumérica**
- Partiendo de la secuencia alfanumérica **no se puede generar de nuevo el texto plano** de entrada
- Es muy difícil que **dos textos** planos tengan la **misma** cadena alfanumérica de salida

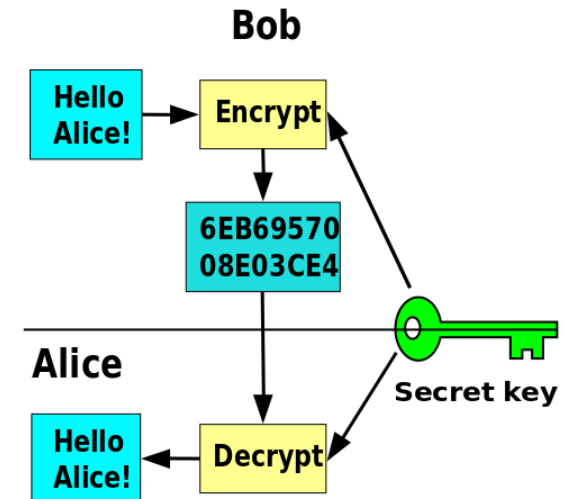
- **Confidencialidad**

- La confidencialidad se consigue típicamente usando técnicas criptográficas de **cifrado de mensajes**
- Tipos de sistemas criptográficos:
  - **Clave secreta** (simétricos)
  - **Clave pública** (asimétricos)



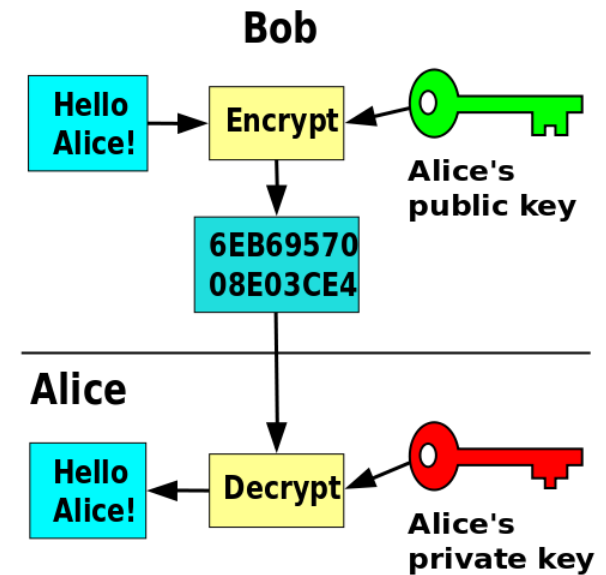
- **Clave secreta**

- En ellos, la **clave de cifrado** y de **descifrado** es la **misma**: es una clave secreta que comparten el emisor y el receptor del mensaje.
- Debido a esta característica son denominados también criptosistemas **simétricos**



- **Clave pública**

- Se distinguen porque cada usuario dispone de dos claves: una **privada**, que debe mantener secreta, y una **pública**, que debe ser conocida por todas las restantes entidades que van a comunicar con ella.
- Se los conoce también como criptosistemas **asimétricos**



- **Certificados digitales**
  - En los sistemas de **clave pública**, un **certificado digital** es un fichero que asocia el **nombre de una entidad** con su **clave pública**
  - El certificado digital es emitido por una **Autoridad de Certificación (CA)**, es decir, una entidad reconocida de confianza o “Tercera Parte de Confianza” (TTP, Trusted Third Party)
  - Los certificados usados en **aplicaciones web** asocian un **dominio web** a su **clave pública**

- **Diferentes algoritmos de cada tipo**

## **Criptosistemas asimétricos**

- RSA (Rivest, Shamir y Adleman)
- Diffie-Hellman
- ElGamal
- Criptografía de curva elíptica

## **Funciones hash**

- SHA (*Secure Hash Algorithm*)
- MD5 (*Message-Digest Algorithm 5*)
- DSA (*Digital Signature Algorithm*)

## **Criptosistemas simétricos**

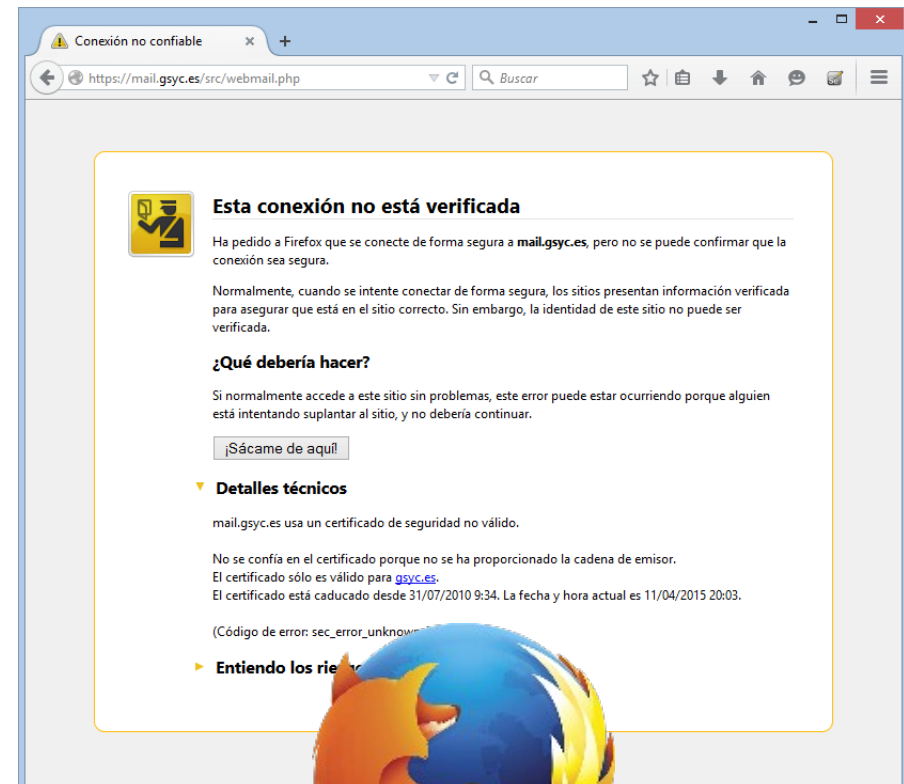
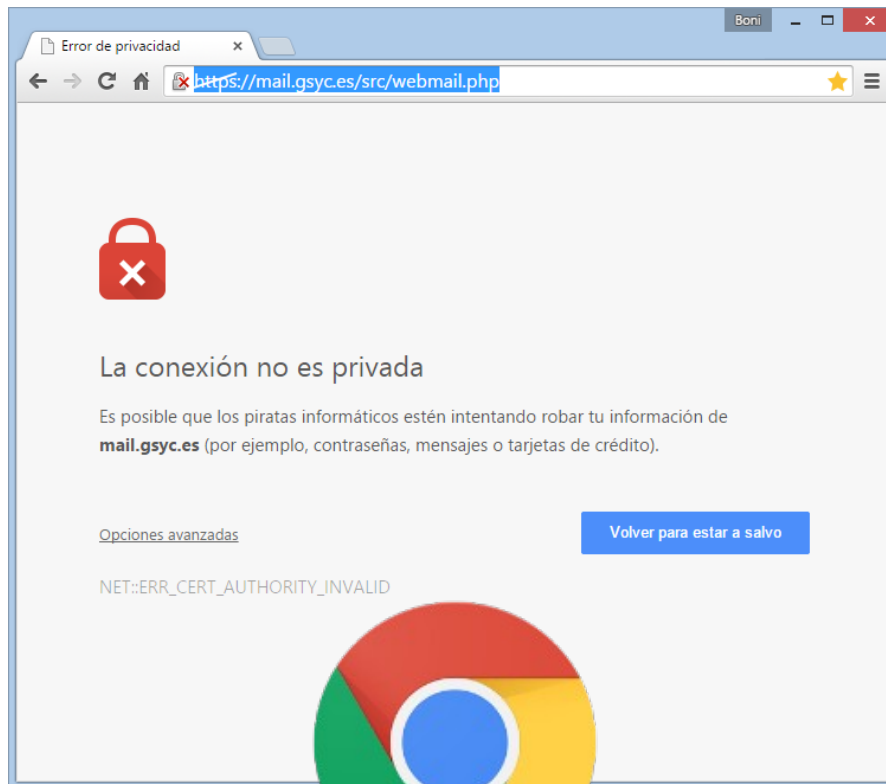
- AES (*Advanced Encryption Standard*)
- ES (*Data Encryption Standard*)
- IDEA (*International Data Encryption Algorithm*)
- 3DES
- RC2, RC4, RC5
- Blowfish

- Introducción
- **Https**
- Spring Security
- API REST para web SPA con seguridad

- **Https** (*Hypertext Transfer Protocol Secure*):  
Versión segura de HTTP
- Con HTTPS se consigue que toda la información que se intercambie un **navegador** web con un **servidor** web esté **cifrada**
- Es decir, un usuario malicioso no podrá entender la información que viaja por la red
- HTTPS utiliza criptografía de **clave pública** y se apoya en el estándar **TLS**

- Los navegadores tienen una **lista de CA** en la que confían
- Si se conectan a un servidor web y presenta un certificado que esté firmado por una CA no reconocida, se muestra un **aviso al usuario**

# Https

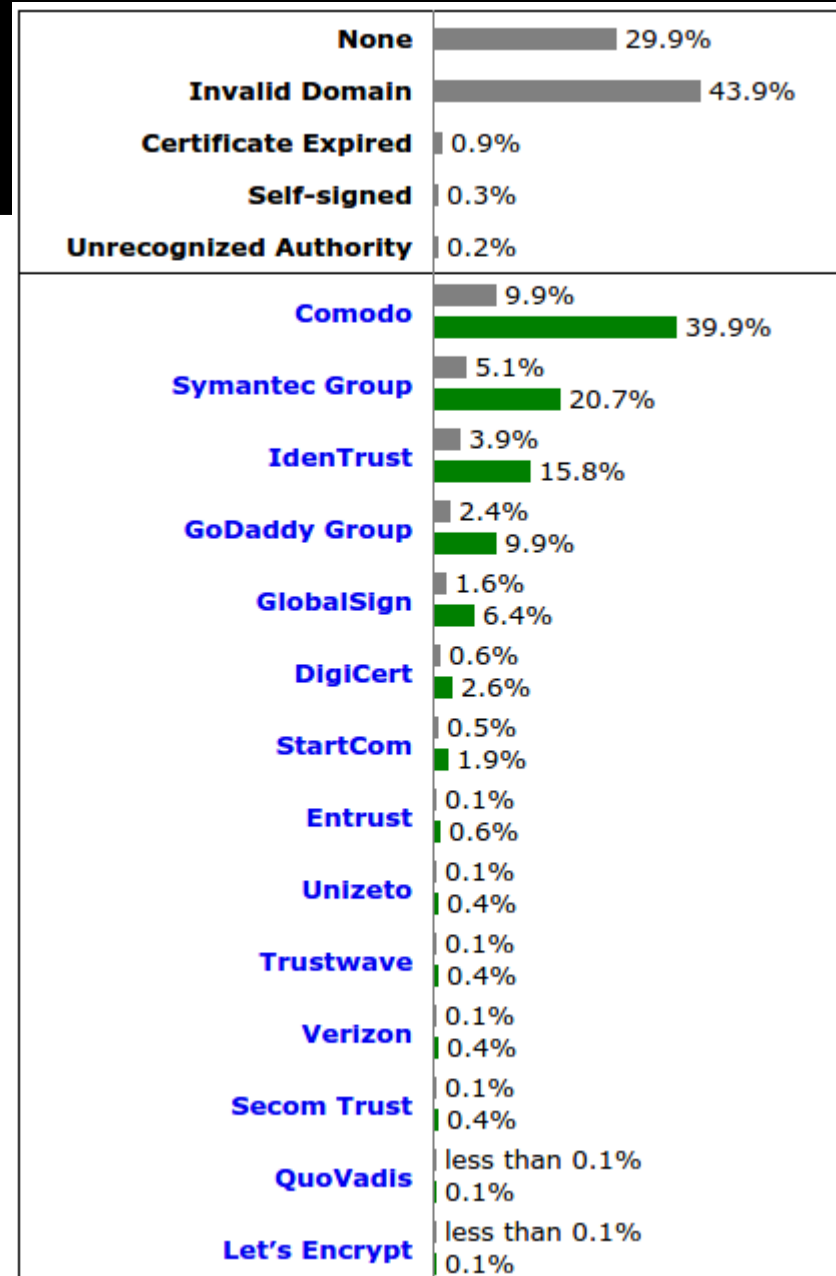
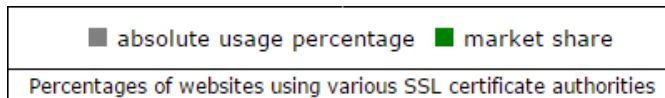




- Un certificado está asociado a un dominio y se puede conseguir de dos formas diferentes:
  - **Comprándolo a una CA:**
    - Puede costar entre **10€ y 1000€** anuales.
    - Existen muchas empresas dedicadas a este negocio
  - **Obteniéndolo de Let's Encrypt:**
    - Entidad sin ánimo de lucro que proporciona certificados de confianza (Let's Encrypt es una CA). Apoyada por Facebook, Mozilla, ...
    - Hasta 20 certificados a la semana, 3 meses de validez
    - Herramientas para obtención y renovación automática
  - **Creándolo uno mismo:**
    - Es gratis
    - Los navegadores mostrarán el **aviso** de entidad no reconocida a los usuarios

# Https

- Estadísticas de uso de CAs



- Introducción
- Https
- **Spring Security**
- API REST para web SPA con seguridad

- Es el proyecto de spring encargado de la **seguridad web**
- Las funcionalidades más importantes:
  - **Autenticación** de usuarios (p.e. con user y pass)
  - **Autorización** de acceso a URLs web y métodos Java
- El servidor web embebido (Tomcat) ofrece soporte de **https**

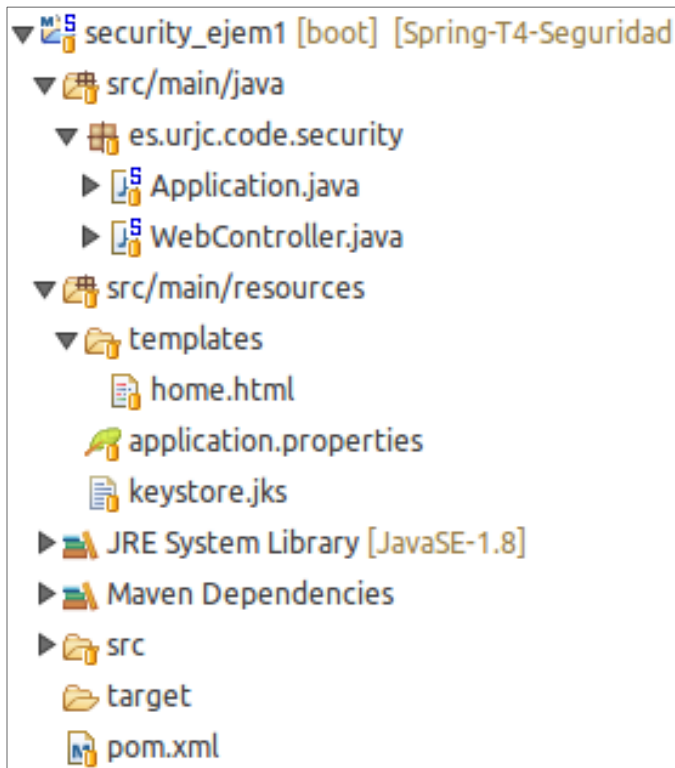
<http://projects.spring.io/spring-security/>

- Spring security permite implementar muchos **esquemas diferentes** de seguridad
- Estudiaremos algunos de los más sencillos
  - **Ejemplo 1:** Comunicación cifrada por https
  - **Ejemplo 2:** Un usuario con credenciales en el código o en el fichero de configuración
  - **Ejemplo 3:** Protección con CSRF
  - **Ejemplo 4:** Diferentes tipos de usuarios (roles)
  - **Ejemplo 5:** Usuarios en BBDD

- 1) Comunicación cifrada con https
  - No es necesario **spring-security** porque esta funcionalidad la ofrece el servidor web Tomcat
  - Usaremos un **certificado autofirmado** que generará un aviso de seguridad en el navegador

- 1) Comunicación cifrada con https
  - Usando un **certificado autofirmado** podríamos sufrir un ataque y nuestros datos podrían ser descifrados
  - Si usamos un **certificado de una CA**, nuestros datos no podrán ser descifrados ni alterados

- 1) Comunicación cifrada con https

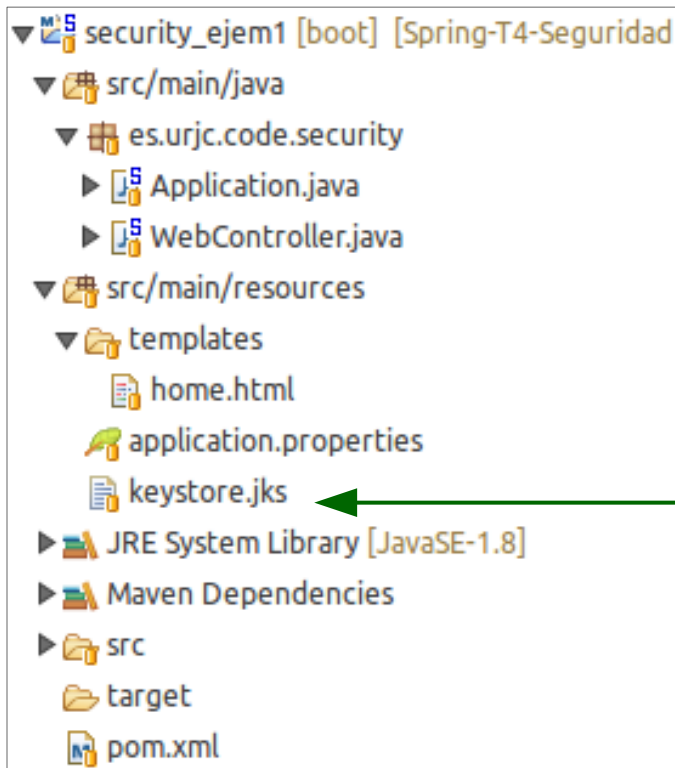


## pom.xml

```
<dependencies>
  <dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-web</artifactId>
  </dependency>
  <dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-mustache</artifactId>
  </dependency>
</dependencies>
```



- 1) Comunicación cifrada con https



application.properties

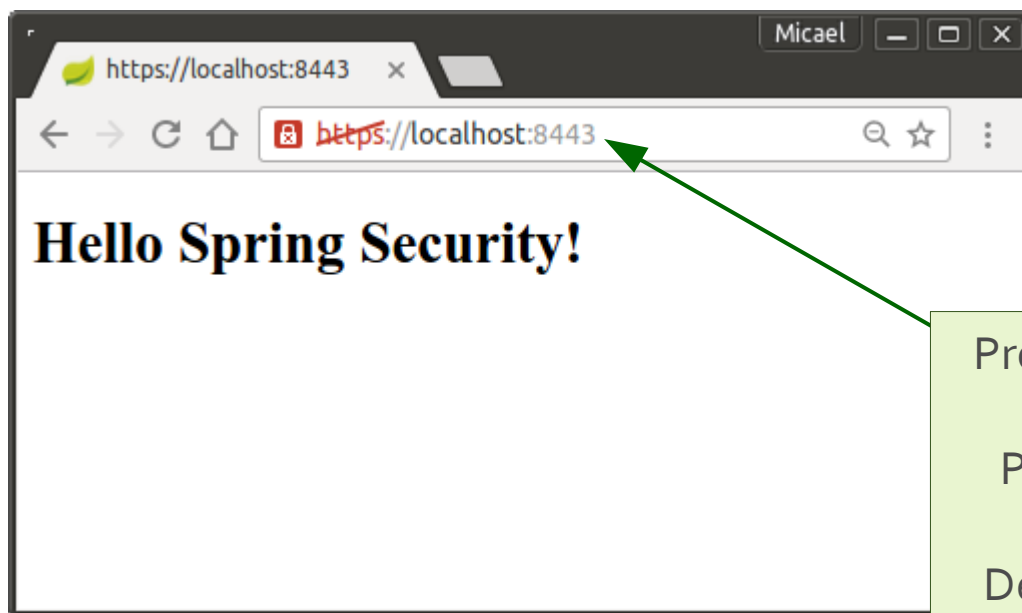
```
server.port = 8443
server.ssl.key-store = classpath:keystore.jks
server.ssl.key-store-password = password
server.ssl.key-password = secret
```

Fichero que contiene el  
certificado autofirmado generado  
con la herramienta del JDK  
**keytool**

- 1) Comunicación cifrada con https

```
$ cd $JAVA_HOME/bin
$ keytool -genkey -keyalg RSA -alias selfsigned -keystore keystore.jks -storepass
password -validity 360 -keysize 2048
¿Cuáles son su nombre y su apellido?
[Unknown]: Micael Gallego
¿Cuál es el nombre de su unidad de organización?
[Unknown]: Code
¿Cuál es el nombre de su organización?
[Unknown]: URJC
¿Cuál es el nombre de su ciudad o localidad?
[Unknown]: Madrid
¿Cuál es el nombre de su estado o provincia?
[Unknown]: Madrid
¿Cuál es el código de país de dos letras de la unidad?
[Unknown]: ES
¿Es correcto CN=Micael Gallego, OU=Code, O=URJC, L=Madrid, ST=Madrid,
C=ES?
[no]: si
Introduzca la contraseña de clave para <selfsigned>
      (INTRO si es la misma contraseña que la del almacén de claves): secret
Volver a escribir la contraseña nueva: secret
```

- 1) Comunicación cifrada con https



Protocolo https (en vez de http)

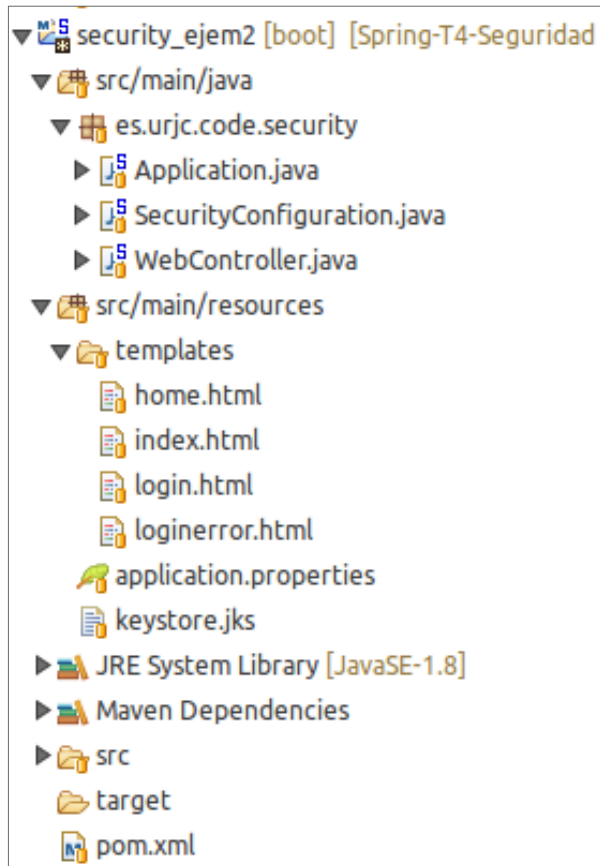
Puerto 8443 (en vez de 8080)

Después de aceptar el aviso de seguridad accedemos a la web

- 2) Usuario con credenciales en código
  - Spring-security **impide** que un usuario pueda **acceder a ciertas páginas** si no se ha autenticado correctamente
  - Si el usuario intenta acceder, se le redirige al **formulario de login**
  - El desarrollador **configura** qué páginas son **públicas** y cuales son **privadas**

<http://projects.spring.io/spring-security/>

- 2) Usuario con credenciales en código



pom.xml

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

Dependencia spring-security

- 2) Usuario con credenciales en código

WebController.java

```
@Controller
public class WebController {


    @GetMapping("/")
    public String index() {
        return "index";
    }

    @GetMapping("/login")
    public String login() {
        return "login";
    }

    @GetMapping("/loginerror")
    public String loginerror() {
        return "loginerror";
    }

    @GetMapping("/home")
    public String home() {
        return "home";
    }
}
```

Controlador que asocia  
URLs a plantillas



# Spring security

ejem2

```
@Configuration
public class SecurityConfiguration extends WebSecurityConfigurerAdapter {

    @Override
    protected void configure(HttpSecurity http) throws Exception {

        // Public pages
        http.authorizeRequests().antMatchers("/").permitAll();
        http.authorizeRequests().antMatchers("/login").permitAll();
        http.authorizeRequests().antMatchers("/loginerror").permitAll();
        http.authorizeRequests().antMatchers("/logout").permitAll();

        // Private pages (all other pages)
        http.authorizeRequests().anyRequest().authenticated();

        // Login form
        http.formLogin().loginPage("/login");
        http.formLogin().usernameParameter("username");
        http.formLogin().passwordParameter("password");
        http.formLogin().defaultSuccessUrl("/home");
        http.formLogin().failureUrl("/loginerror");

        // Logout
        http.logout().logoutUrl("/logout");
        http.logout().logoutSuccessUrl("/");

        // Disable CSRF at the moment
        http.csrf().disable();

    }
}
```

URLs públicas

URLs privadas (las demás)

Configuración del formulario de login

Configuración de la página de logout

Deshabilitamos CSRF


- 2) Usuario con credenciales en código

```
@Configuration
public class SecurityConfiguration extends WebSecurityConfigurerAdapter {

    ...

    @Override
    protected void configure(AuthenticationManagerBuilder auth) throws Exception {

        // User
        auth.inMemoryAuthentication()
            .withUser("user").password("pass").roles("USER");
    }
}
```



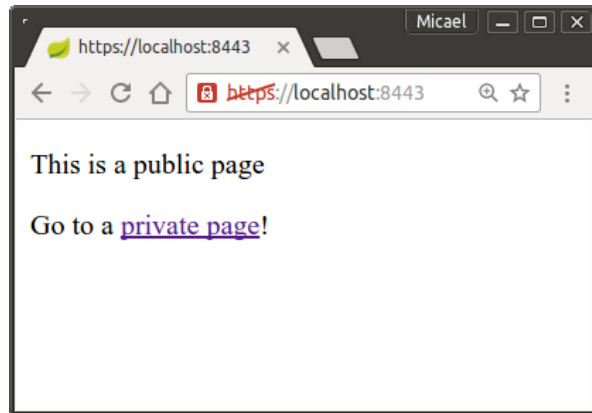
Datos del único  
usuario válido en la  
aplicación



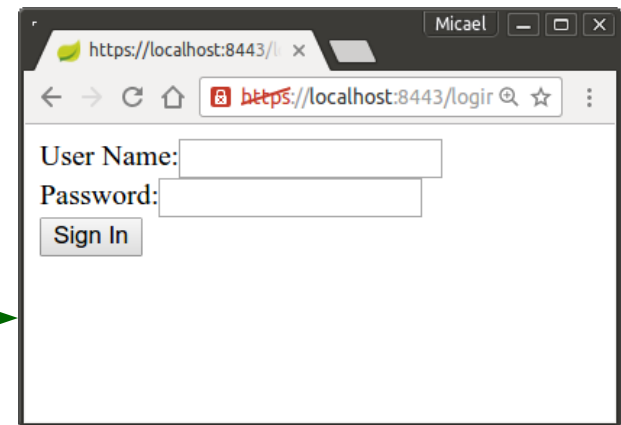
# Spring security

- 2) Usuario con credenciales en código

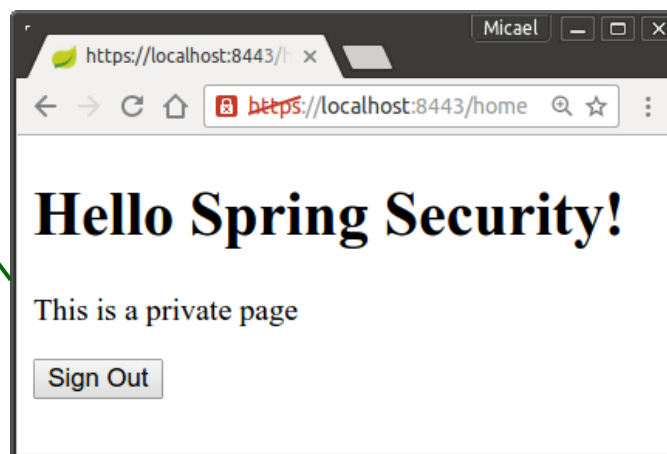
ejem2



/home es una URL privada, así que se redirige la navegación al formulario de login



Sign Out



Con las credenciales correctas vamos a /home

- 3) Protección con CSRF
  - **Cross Site Request Forgery** es un tipo de ataque en el que una página web intenta hacer una petición a otra web en la que estás logueado (correo, banco, etc...)
  - La forma de evitarlo es generar un **token** por cada **formulario** y verificar que el token es válido al procesar los datos del formulario

<http://docs.spring.io/autorepo/docs/spring-security/current/reference/html/csrf.html>


[http://en.wikipedia.org/wiki/Cross-site\\_request\\_forgery](http://en.wikipedia.org/wiki/Cross-site_request_forgery)

- 3) Protección con CSRF

- Cada formulario en una web con protección CSRF debería ser similar a este

```
<!DOCTYPE html>
<html>
<body>
  <h1>Hello Spring Security!</h1>
  <p>This is a private page</p>
  <form method="post" action="/logout">
    <input type="submit" value="Sign Out" />
    <input type="hidden" name="_csrf"
      value="c54a70a7-1586-4dc3-8e64-4fac09625ce2" />
  </form>
</body>
</html>
```

Token CSRF  
generado por  
spring-security



- **3) Protección con CSRF**

- Para usar CSRF con Mustache tenemos que cargar en el modelo el token desde la request

```
@RequestMapping("/login")
public String login(Model model, HttpServletRequest request) {

    CsrfToken token = (CsrfToken) request.getAttribute("_csrf");
    model.addAttribute("token", token.getToken());

    return "login";
}
```

- **3) Protección con CSRF**
  - Y generar el formulario usando ese token

```
<!DOCTYPE html>
<html>
<body>
  <h1>Hello Spring Security!</h1>
  <p>This is a private page</p>
  <form action="/logout" method="post">
    <input type="submit" value="Sign Out" />
    <input type="hidden" name="_csrf" value="{{token}}"/>
  </form>
</body>
</html>
```

- 3) Protección con CSRF
  - Pasar el token al modelo en cada método del controlador es **repetitivo**
  - Para evitarlo, podemos implementar un **Handler** que se ejecutará después de nuestro método

- 3) Protección con CSRF

```
@Configuration
public class CSRFHandlerConfiguration extends WebMvcConfigurerAdapter {

    @Override
    public void addInterceptors(InterceptorRegistry registry) {
        registry.addInterceptor(new CSRFHandlerInterceptor());
    }
}

class CSRFHandlerInterceptor extends HandlerInterceptorAdapter {

    @Override
    public void postHandle(final HttpServletRequest request,
        final HttpServletResponse response, final Object handler,
        final ModelAndView modelAndView) throws Exception {

        CsrfToken token = (CsrfToken) request.getAttribute("_csrf");
        modelAndView.addObject("token", token.getToken());
    }
}
```

Activación  
del Handler

Handler

- **3) Protección con CSRF**

- **Mustache** es un sistema de plantillas muy básico.
- Otros sistemas de plantillas más avanzados como **Thymeleaf** se integran con Spring y generan automáticamente el campo para el token CSRF en los formularios (sin necesidad de meter el token en el modelo)



- 3) Protección con CSRF
  - Formularios con Thymeleaf

```
<!DOCTYPE html>
<html xmlns:th="http://www.thymeleaf.org">
<body>
<h1>Hello Spring Security!</h1>
<form th:action="@{/logout}" method="post">
  <input type="submit" value="Sign Out" />
</form>
</body>
</html>
```



```
<!DOCTYPE html>
<html>
<body>
  <h1>Hello Spring Security!</h1>
  <form method="post" action="/logout">
    <input type="submit" value="Sign Out" />
    <input type="hidden" name="_csrf"
      value="c54a70a7-1586-4dc3-8e64" />
  </form>
</body>
</html>
```

- 4) Diferentes tipos de usuarios (roles)
  - Damos de alta dos usuarios, uno con rol USER y otro ADMIN

```
@Configuration
public class SecurityConfiguration extends WebSecurityConfigurerAdapter {

    @Override
    protected void configure(AuthenticationManagerBuilder auth)
        throws Exception {

        // Users
        auth.inMemoryAuthentication().withUser("user").password("pass")
            .roles("USER");

        auth.inMemoryAuthentication().withUser("admin").password("adminpass")
            .roles("USER", "ADMIN");

    }
}
```

- 4) Diferentes tipos de usuarios (roles)
  - Configuramos las páginas que puede ver cada tipo de usuario

```
@Configuration
public class SecurityConfiguration extends WebSecurityConfigurerAdapter {

    @Override
    protected void configure(HttpSecurity http) throws Exception {

        // Public pages
        ...
        // Private pages (all other pages)
        http.authorizeRequests().antMatchers("/home").hasAnyRole("USER");
        http.authorizeRequests().antMatchers("/admin").hasAnyRole("ADMIN");

        // Login form
        ...
        // Logout
        ...
    }
}
```

- 4) Diferentes tipos de usuarios (roles)
  - En el controlador podemos saber el rol del usuario que se ha logueado

```
@RequestMapping("/home")  
public String home(Model model, HttpServletRequest request) {  
  
    model.addAttribute("admin", request.isUserInRole("ADMIN"));  
    return "home";  
}
```

```
<h1>Hello Spring Security!</h1>  
  
{{#admin}}  
<a href="/admin">Admin Page</a>  
{{/admin}}
```

- **5) Usuarios en BBDD**

- Hasta ahora todos los **usuarios** eran **fijos** y sus credenciales están en el **código fuente**
- Habitualmente los **usuarios** están en la **BD** y se pueden **añadir, borrar, modificar**, etc.

- **5) Usuarios en BBDD**
  - Para guardar usuarios en la BD creamos una **entidad WebUser** y un **WebUserRepository**
  - Desde el punto de vista de **Spring Security** tenemos que proporcionar un **proveedor de autenticación basado en BD**
  - En ese proveedor, cada vez que un usuario se quiera autenticar, comprobamos **si está en la BD** y si la **contraseña es la correcta**

- 5) Usuarios en BBDD

```
@Entity
public class User {


    @Id
    @GeneratedValue(strategy = GenerationType.AUTO)
    private Long id;

    private String name;
    private String passwordHash;

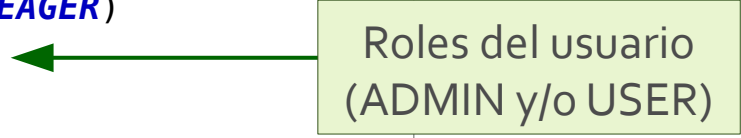
    @ElementCollection(fetch = FetchType.EAGER)
    private List<String> roles;

    //Constructor, getters and setters
}
```

Contraseña cifrada con  
una función hash



Roles del usuario  
(ADMIN y/o USER)



```
public interface UserRepository extends CrudRepository<User, Long> {

    User findByName(String name);
}
```

- 5) Usuarios en BBDD

```
@Component
public class DatabaseUsersLoader {

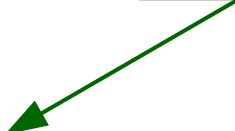
    @Autowired
    private UserRepository userRepository;

    @PostConstruct
    private void initDatabase() {

        userRepository.save(
            new User("user", "pass", "ROLE_USER"));

        userRepository.save(
            new User("admin", "adminpass", "ROLE_USER", "ROLE_ADMIN"));
    }
}
```

Guardamos en la BBDD  
usuarios de ejemplo





- 5) Usuarios en BBDD

```
@Configuration
public class SecurityConfiguration extends WebSecurityConfigurerAdapter {


    @Autowired
    public UserRepositoryAuthenticationProvider authenticationProvider;

    @Override
    protected void configure(HttpSecurity http) throws Exception {...}


    @Override
    protected void configure(AuthenticationManagerBuilder auth)
        throws Exception {

        // Database authentication provider
        auth.authenticationProvider(authenticationProvider);
    }
}
```

El proveedor se inyecta  
en la configuración de  
seguridad



Configuramos el  
proveedor en  
SpringSecurity



# Spring security

ejem5

```
@Component
public class UserRepositoryAuthenticationProvider implements AuthenticationProvider {

    @Autowired
    private UserRepository userRepository;

    @Override
    public Authentication authenticate(Authentication auth) throws AuthenticationException {

        User user = userRepository.findByName(auth.getName());

        if (user == null) {
            throw new BadCredentialsException("User not found");
        }

        String password = (String) auth.getCredentials();
        if (!new BCryptPasswordEncoder().matches(password, user.getPasswordHash())) {
            throw new BadCredentialsException("Wrong password");
        }

        List<GrantedAuthority> roles = new ArrayList<>();
        for (String role : user.getRoles()) {
            roles.add(new SimpleGrantedAuthority(role));
        }

        return new UsernamePasswordAuthenticationToken(user.getName(), password, roles);
    }
    ...
}
```

Cargamos el usuario de la BBDD

Si existe y la contraseña es correcta devolvemos un objeto con los roles

- Introducción
- Https
- Spring Security
- **APIs REST seguras**

# APIs REST seguras

- Existen muchas **formas diferentes** de hacer segura una **API REST**
  - Basic Authentication (desaconsejada)
  - OAuth1a
  - OAuth2 (completamente diferente a OAuth1)
  - JWT

# APIs REST seguras

- Una API REST debe responder con diferentes códigos HTTP que una aplicación MVC:
  - Login correcto: **200 ok** vs 301 Moved permanently
  - Acceso requiere credenciales: **401 Unauthorized** (usuario no autenticado)
  - Acceso con credenciales del usuario no válido: **403 Forbidden** (usuario no autorizado)

# APIs REST seguras

- <http://www.infoq.com/presentations/spring-security-rest-api>
- <http://www.infoq.com/presentations/spring-security-2015>
- <http://www.infoq.com/presentations/spring-4-security>
- <https://spring.io/guides/gs/authenticating-ldap/>
- <http://www.baeldung.com/securing-a-restful-web-service-with-spring-security>
- <http://es.slideshare.net/stormpath/secure-your-rest-api-the-right-way>

- **No** vamos a presentar las diferentes **alternativas** de seguridad de **APIs REST** con **Spring**
- Vamos a describir una forma de **securizar** una API REST diseñada para una **web SPA** con **Angular 2**

- **Características**

- Frontend con **Angular 2** y **bootstrap**
- Backend con **Spring Boot** y **BD H2**
- Gestión de libros **CRUD**
- Acceso público: **Consulta** de libros
- Acceso restringido (autenticado):
  - **User:** Crear o editar libros
  - **Admin:** Borrar libros



- **Características de seguridad**
  - Comunicación cifrada con **HTTPS**
  - **Autenticación** mediante **HTTP Basic Auth**
  - **Autorización** basada en **URLs**
  - Gestión de sesión: **Cookies** y **HttpSession**
  - No se usa protección para **CSRF**

- **Seguridad en Backend**

- **pom.xml:** Dependencia spring-security
- **SecurityConfig:** Configuración de seguridad de la aplicación
- **LoginController:** Controlador REST para login y logout desde la SPA
- **UserRepositoryAuthProvider:** Clase que busca a los usuarios en la BBDD cuando se quieren autenticar en la app

- **Seguridad en Backend**

- **User:** Entidad del usuario en la BBDD. Nombre, password, roles y demás info que se quiera incluir
- **UserRepository:** Repositorio para gestionar las entidades User. Se pueden incorporar métodos para consultas
- **UserComponent:** Componente que guarda la información en memoria sobre el usuario. Tiene una referencia al User (de la BBDD)

- **Seguridad en Frontend**

- **login.service.ts:** Método para login y logout. Atributos para consultar isLoggedIn e isAdmin.
- **login.component.ts:** Formulario de login. Si hay usuario logueado muestra su nombre y botón para logout
- **Otros componentes:** Se ocultan las operaciones no permitidas si no está autenticado o no está autorizado (los no admin no pueden borrar)

- **Limitaciones**

- No se permite crear/editar/borrar usuarios
- La autorización sólo se basa en URL/método. En una aplicación más compleja hay que comprobar manualmente si el usuario autenticado tiene privilegios para acceder a los datos que solicita

- **Ejemplos JWT**

<https://www.adictosaltrabajo.com/2017/09/25/securizar-un-api-rest-utilizando-json-web-tokens/>

<https://dzone.com/articles/spring-boot-security-json-web-tokenjwt-hello-world>

- **Ejemplo OAUTH2**

- Este ejemplo es más complejo y consta de varios servicios
  - Servicios Oauth con Spring
  - Servicio REST
  - App SPA Angular

<https://www.baeldung.com/rest-api-spring-oauth2-angular>