Securizar un API REST utilizando JSON Web Tokens

Por **Álvaro Javier Morgan** - 25 septiembre, 2017

Índice de contenidos

1. Introducción

En este tutorial veremos cómo securizar un API REST empleando JSON Web Tokens (JWT). Para este tutorial utilizaremos un API muy simple donde activaremos Spring Security con la configuración adecuada e implementaremos las clases necesarias para el uso de JWT.

En anteriores tutoriales vimos con securizar un API REST utilizando Node.js y JWT, en esta ocasion utilizaremos Spring Boot ya que nos permite desarrollar rápidamente API REST con el mínimo código y por tanto con el menor número de errores 😉

Disponéis de un tutorial de Natalia Roales en el portal sobre proyectos con Spring Boot y os dejamos el código fuente de este tutorial en github para vuestra consulta.

2. Entorno

El tutorial está escrito usando el siguiente entorno:

- Hardware: Portátil Mac Book Pro 15" (2,5 Ghz Intel Core i7, 16 GB DDR3)
- Sistema Operativo: Mac OS Sierra 10.12.6
- Entorno de desarrollo: Spring Tool Suite 3.9.0

3. JSON Web Token

JSON Web Token (JWT) es un estándar abierto (RFC 7519) que define un modo compacto y autónomo para transmitir de forma segura la información entre las partes como un objeto JSON. Esta información puede ser verificada y es confiable porque está firmada digitalmente.

Los JWT se pueden firmar usando un secreto (con el algoritmo HMAC) o utilizando un par de claves públicas / privadas usando RSA.

3.1. Ventajas de los tokens frente a las cookies

Quizás la mayor ventaja de los tokens sobre las cookies es el hecho de que no tenga estado (stateless). El backend no necesita mantener un registro de los tokens. Cada token es compacto y auto contenido. Contiene todos los datos necesarios para comprobar su validez, así como la información del usuario para las diferentes peticiones.

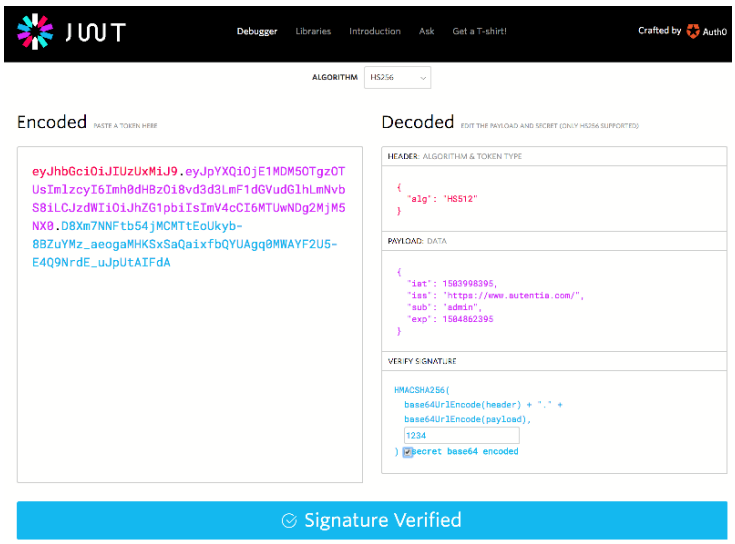
El único trabajo del servidor consiste en firmar tokens al iniciar la sesión y verificar que los tokens intercambiados sean válidos. Esta característica permite la escalabilidad inmediata ya que las peticiones no dependen unas de otras. De esta forma se pueden tramitar en diferentes servidores de forma autónoma.

Las cookies funcionan bien con dominios y subdominios únicos, pero cuando se trata de administrar cookies en diferentes dominios, puede volverse complejo. El enfoque basado en tokens con Cross Origin Resource Sharing (CORS) habilitado hace trivial exponer las API a diferentes servicios y dominios.

Con un enfoque basado en cookies, simplemente se almacena el ID de sesión en una cookie. JWT por otro lado permiten almacenar cualquier tipo de metadatos, siempre y cuando sea un JSON válido.

Si os preguntáis por el rendimiento, cuando se utiliza la autenticación basada en cookies, el backend tiene que hacer una búsqueda habitualmente una base de datos para recuperar la información del usuario, esto seguramente supera el tiempo que pueda tomar la decodificación de un token. Además, puesto que se pueden almacenar datos adicionales dentro del JWT, por ejemplo, permisos de usuario, puede ahorrarse llamadas adicionales para la búsqueda de esta información.

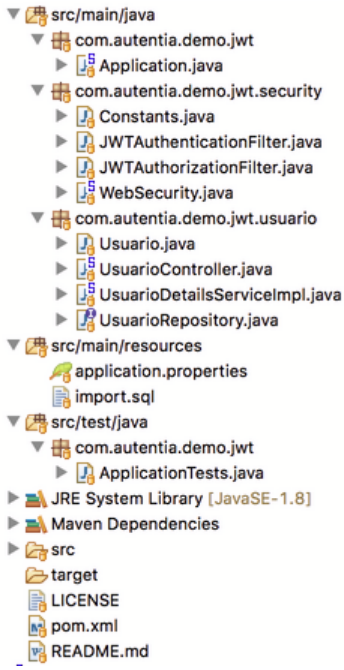
Recordad que el token habitualmente va firmado pero no va cifrado. En la web de jwt.io disponéis de un depurador que permite consultar y comprobar la validez del mismo.



Después de tanta teoría vamos a lo interesante.

4. Estructura del proyecto

Nuestro proyecto a nivel de Maven está configurado para ser un proyecto Spring Boot de tipo web, que utiliza Spring Security, JPA y HSQLDB (podéis consultar el pom.xml). Veamos las partes más importantes del ejemplo, consta de un controlador para acceso al API REST, el acceso a la capa de datos por JPA y los beans de dominio, muy simple. Para esta demostración utilizaremos la base de datos en memoria HSQLDB (HyperSQL Database).



A nivel de controlador se disponen de métodos para crear un usuario, consultar todos o consultar uno concreto. Para evitar almacenar las password en plano, aplicamos una función de Hash basada en el cifrado Blowfish (BCrypt). Recordad que el uso de MD5 para almacenar las password no se recomienda, utilizad algoritmos más modernos.

UsuarioController.java

Java

```java
package com.autentia.demo.jwt.usuario;

import java.util.List;

import org.springframework.security.crypto.bcrypt.BCryptPasswordEncoder;
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.PathVariable;
import org.springframework.web.bind.annotation.PostMapping;
import org.springframework.web.bind.annotation.RequestBody;
import org.springframework.web.bind.annotation.RestController;

@RestController
public class UsuarioController {

    private UsuarioRepository usuarioRepository;

    private BCryptPasswordEncoder bCryptPasswordEncoder;

    public UsuarioController(UsuarioRepository usuarioRepository, BCryptPasswordEncoder
        this.usuarioRepository = usuarioRepository;
        this.bCryptPasswordEncoder = bCryptPasswordEncoder;
```

```
    }

    @PostMapping("/users/")
    public void saveUsuario(@RequestBody Usuario user) {
        user.setPassword(bCryptPasswordEncoder.encode(user.getPassword()));
        usuarioRepository.save(user);
    }

    @GetMapping("/users/")
    public List<Usuario> getAllUsuarios() {
        return usuarioRepository.findAll();
    }

    @GetMapping("/users/{username}")
    public Usuario getUsuario(@PathVariable String username) {
        return usuarioRepository.findByUsername(username);
    }
}
```

5. Spring Security

Gracias a Spring Security podemos incorporar mecanismos potentes para proteger nuestras aplicaciones utilizando una cantidad mínima de código.

En nuestro caso indicamos a Spring Security que proteja todas las URLs excepto la URL de login, así mismo, declaramos las implementaciones que utilizaremos para realizar la autenticación y autorización.

WebSecurity.java

Java

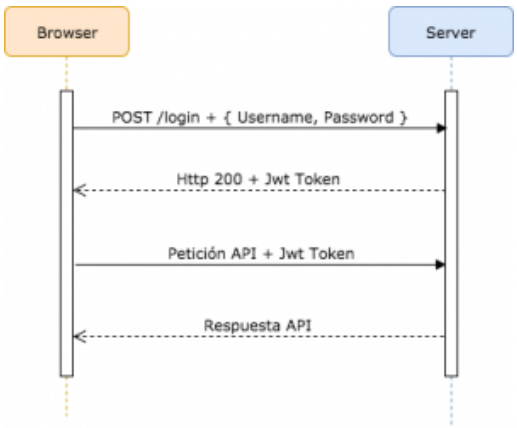
```
package com.autentia.demo.jwt.security;

import static com.autentia.demo.jwt.security.Constants.LOGIN_URL;

import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import org.springframework.http.HttpMethod;
import org.springframework.security.config.annotation.authentication.builders.Authentico
import org.springframework.security.config.annotation.web.builders.HttpSecurity;
import org.springframework.security.config.annotation.web.configuration.EnableWebSecurit
import org.springframework.security.config.annotation.web.configuration.WebSecurityConfi
import org.springframework.security.config.http.SessionCreationPolicy;
import org.springframework.security.core.userdetails.UserDetailsService;
import org.springframework.security.crypto.bcrypt.BCryptPasswordEncoder;
import org.springframework.web.cors.CorsConfiguration;
import org.springframework.web.cors.CorsConfigurationSource;
import org.springframework.web.cors.UrlBasedCorsConfigurationSource;

@Configuration
@EnableWebSecurity
public class WebSecurity extends WebSecurityConfigurerAdapter {

    private UserDetailsService userDetailsService;

    public WebSecurity(UserDetailsService userDetailsService) {
        this.userDetailsService = userDetailsService;
    }

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

    @Override
    protected void configure(HttpSecurity httpSecurity) throws Exception {
        /*
         * 1. Se desactiva el uso de cookies
```

```
38              * 2. Se activa la configuración CORS con los valores por defecto
39              * 3. Se desactiva el filtro CSRF
40              * 4. Se indica que el login no requiere autenticación
41              * 5. Se indica que el resto de URLs esten securizadas
42              */
43             httpSecurity
44                 .sessionManagement().sessionCreationPolicy(SessionCreationPolicy.STATELESS).
45                 .cors().and()
46                 .csrf().disable()
47                 .authorizeRequests().antMatchers(HttpMethod.POST, LOGIN_URL).permitAll()
48                 .anyRequest().authenticated().and()
49                     .addFilter(new JWTAuthenticationFilter(authenticationManager()))
50                     .addFilter(new JWTAuthorizationFilter(authenticationManager()));
51         }
52 
53         @Override
54         public void configure(AuthenticationManagerBuilder auth) throws Exception {
55             // Se define la clase que recupera los usuarios y el algoritmo para procesar las
56             auth.userDetailsService(userDetailsService).passwordEncoder(bCryptPasswordEncode
57         }
58 
59         @Bean
60         CorsConfigurationSource corsConfigurationSource() {
61             final UrlBasedCorsConfigurationSource source = new UrlBasedCorsConfigurationSour
62             source.registerCorsConfiguration("/**", new CorsConfiguration().applyPermitDefau
63             return source;
64         }
65     }
```

Podemos observar que se ajusta la configuración para CORS y se desactiva el filtro de Cross-site request forgery (CSRF). Esto nos permite habilitar el API para cualquier dominio, esta es una de las grandes ventajas del uso de JWT.

6. Implementación

En el siguiente diagrama podéis observar el flujo habitual de una aplicación securizada. Si lo comparamos al flujo seguido en la autenticación vía cookies, es muy similar.



Por fin llegamos a la lógica de negocio a utilizar para autenticar y autorizar nuestras peticiones. Para simplificar este tutorial, se verificará únicamente que exista el usuario y password en nuestra base de datos. Se podría incorporar un modelo más complejo incorporando permisos y roles pero se aleja del objetivo de este tutorial. Si os interesa estos temas, podéis consultar su manejo en la documentación de Spring Security.

6.1. Autenticación

Haciendo uso de las clases proporcionadas por Spring Security, extendemos su comportamiento para reflejar nuestras necesidades. Se verifica que las credencias proporcionadas son válidas y se genera el JWT.

JWTAuthenticationFilter.java

Java

```java
package com.autentia.demo.jwt.security;

import static com.autentia.demo.jwt.security.Constants.HEADER_AUTHORIZACION_KEY;
import static com.autentia.demo.jwt.security.Constants.ISSUER_INFO;
import static com.autentia.demo.jwt.security.Constants.SUPER_SECRET_KEY;
import static com.autentia.demo.jwt.security.Constants.TOKEN_BEARER_PREFIX;
import static com.autentia.demo.jwt.security.Constants.TOKEN_EXPIRATION_TIME;

import java.io.IOException;
import java.util.ArrayList;
import java.util.Date;

import javax.servlet.FilterChain;
import javax.servlet.ServletException;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;

import org.springframework.security.authentication.AuthenticationManager;
import org.springframework.security.authentication.UsernamePasswordAuthenticationToken;
import org.springframework.security.core.Authentication;
import org.springframework.security.core.AuthenticationException;
import org.springframework.security.core.userdetails.User;
import org.springframework.security.web.authentication.UsernamePasswordAuthenticationFil

import com.autentia.demo.jwt.usuario.Usuario;
import com.fasterxml.jackson.databind.ObjectMapper;

import io.jsonwebtoken.Jwts;
import io.jsonwebtoken.SignatureAlgorithm;

public class JWTAuthenticationFilter extends UsernamePasswordAuthenticationFilter {

    private AuthenticationManager authenticationManager;

    public JWTAuthenticationFilter(AuthenticationManager authenticationManager) {
        this.authenticationManager = authenticationManager;
    }

    @Override
    public Authentication attemptAuthentication(HttpServletRequest request, HttpServletR
            throws AuthenticationException {
        try {
            Usuario credenciales = new ObjectMapper().readValue(request.getInputStream()

            return authenticationManager.authenticate(new UsernamePasswordAuthenticatior
                    credenciales.getUsername(), credenciales.getPassword(), new ArrayLis
        } catch (IOException e) {
            throw new RuntimeException(e);
        }
    }

    @Override
    protected void successfulAuthentication(HttpServletRequest request, HttpServletRespo
            Authentication auth) throws IOException, ServletException {

        String token = Jwts.builder().setIssuedAt(new Date()).setIssuer(ISSUER_INFO)
                .setSubject(((User)auth.getPrincipal()).getUsername())
                .setExpiration(new Date(System.currentTimeMillis() + TOKEN_EXPIRATION_TI
                .signWith(SignatureAlgorithm.HS512, SUPER_SECRET_KEY).compact();
        response.addHeader(HEADER_AUTHORIZACION_KEY, TOKEN_BEARER_PREFIX + " " + token);
    }
}
```

No hay obligación de devolver el token en la cabecera ni con una clave concreta pero se recomienda seguir los estándares utilizados en la actualidad (RFC 2616, RFC 6750). Lo

habitual es devolverlo en la cabecera HTTP utilizando la clave "Authorization" e indicando que el valor es un token "**Bearer** " + token

Este token lo deberá conservar vuestro cliente web en su localstorage y remitirlo en las peticiones posteriores que se hagan al API.

6.2. Autorización

La clase responsable de la autorización verifica la cabecera en busca de un token, se verifica el token y se extrae la información del mismo para establecer la identidad del usuario dentro del contexto de seguridad de la aplicación. No se requieren accesos adicionales a BD ya que al estar firmado digitalmente si hay alguna alteración en el token se corrompe.

JWTAuthenticationFilter.java

Java

```java
package com.autentia.demo.jwt.security;

import static com.autentia.demo.jwt.security.Constants.HEADER_AUTHORIZACION_KEY;
import static com.autentia.demo.jwt.security.Constants.SUPER_SECRET_KEY;
import static com.autentia.demo.jwt.security.Constants.TOKEN_BEARER_PREFIX;

import java.io.IOException;
import java.util.ArrayList;

import javax.servlet.FilterChain;
import javax.servlet.ServletException;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;

import org.springframework.security.authentication.AuthenticationManager;
import org.springframework.security.authentication.UsernamePasswordAuthenticationToken;
import org.springframework.security.core.context.SecurityContextHolder;
import org.springframework.security.web.authentication.www.BasicAuthenticationFilter;

import io.jsonwebtoken.Jwts;

public class JWTAuthorizationFilter extends BasicAuthenticationFilter {

    public JWTAuthorizationFilter(AuthenticationManager authManager) {
        super(authManager);
    }

    @Override
    protected void doFilterInternal(HttpServletRequest req, HttpServletResponse res, Fil
            throws IOException, ServletException {
        String header = req.getHeader(HEADER_AUTHORIZACION_KEY);
        if (header == null || !header.startsWith(TOKEN_BEARER_PREFIX)) {
            chain.doFilter(req, res);
            return;
        }
        UsernamePasswordAuthenticationToken authentication = getAuthentication(req);
        SecurityContextHolder.getContext().setAuthentication(authentication);
        chain.doFilter(req, res);
    }

    private UsernamePasswordAuthenticationToken getAuthentication(HttpServletRequest rec
        String token = request.getHeader(HEADER_AUTHORIZACION_KEY);
        if (token != null) {
            // Se procesa el token y se recupera el usuario.
            String user = Jwts.parser()
                        .setSigningKey(SUPER_SECRET_KEY)
                        .parseClaimsJws(token.replace(TOKEN_BEARER_PREFIX, ""))
                        .getBody()
                        .getSubject();

            if (user != null) {
```

```
52                      return new UsernamePasswordAuthenticationToken(user, null, new ArrayList
53                  }
54                  return null;
55              }
56              return null;
57          }
58      }
```

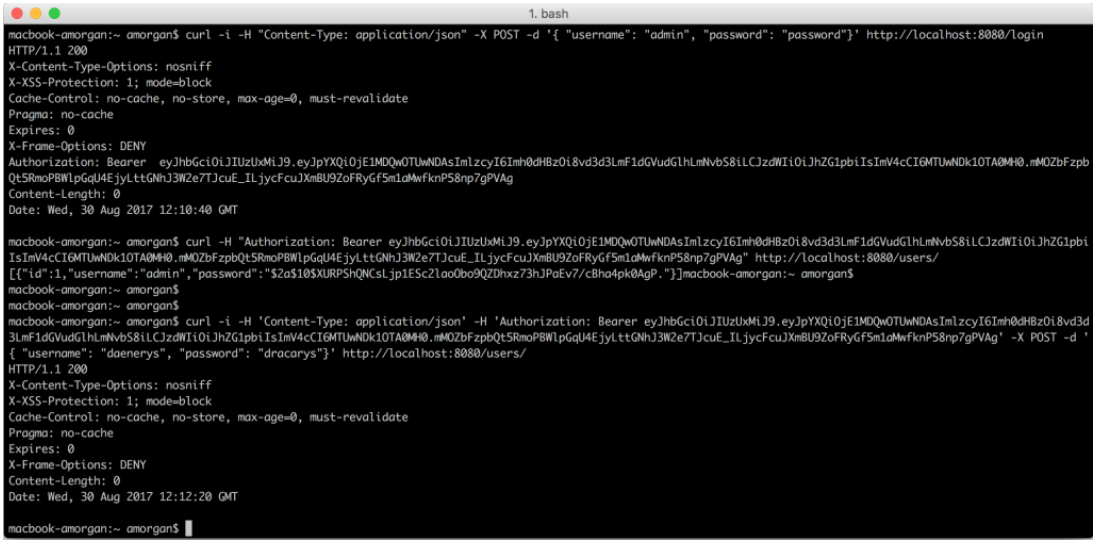
7. Probando...probando

Para las pruebas seguiremos el flujo que vimos previamente, primero, se invoca al login para recuperar el token y posteriormente invocaremos las llamadas al API utilizando el token obtenido.

Shell

```
1   #Se lanza una petición de login
2   curl -i -H "Content-Type: application/json" -X POST -d '{ "username": "admin", "password"
3
4   # Recuperamos los usuarios dados de alta
5   curl -H "Authorization: Bearer xxx.yyy.zzz" http://localhost:8080/users/
6
7   # Damos de alta un nuevo usuario
8   curl -i -H 'Content-Type: application/json' -H 'Authorization: Bearer xxx.yyy.zzz' -X POS
```

Una vez arrancamos la aplicación y lanzamos los comandos obtendremos algo parecido a la imagen adjunta. Si intentamos invocar alguna URL sin el token obtendremos un código de error HTTP 403.



8. Conclusiones

La securización de API es un tema muy extenso y hemos visto una pequeña parte en este tutorial. Como habréis podido observar Spring nos facilita la incorporación de JWT a nuestras APIs gracias a su “magia”. En el caso que no dispongáis de esta posibilidad, os animo a consultar los frameworks y librerías existentes ya que cada vez está más extendido el uso de los tokens para diferentes lenguajes.

Espero que os haya servido

9. Referencias

- Código fuente completo en github.
- Documentación y librerías sobre JWT.
- Tutorial sobre securización de un API REST con node.js y JWT.
- Tutorial de Natalia Roales sobre Spring Boot.
- Documentación sobre Spring Boot.
- Documentación sobre Spring Security.
- Documentación sobre Cross-origin resource sharing (CORS).
- Documentación sobre Cross-Site Request Forgery.

Álvaro Javier Morgan

Consultor tecnológico de desarrollo de proyectos informáticos.

Puedes encontrarme en Autentia: Ofrecemos servicios de soporte a desarrollo, factoría y formación.

Somos expertos en Java/Java EE.