# JSON WEB TOKENS

# +

# SPRING SECURITY

Why should we use JWT and how

# What is JSON Web Token ?

# What is JSON Web Token?

- JSON Web Token (JWT) is an open standard (RFC 7519) that defines a compact and self-contained way for securely transmitting information between parties as a JSON object

- This information can be verified and trusted because it is digitally signed.

- JWTs can be signed using a secret (with the **HMAC** algorithm) or a public/private key pair using **RSA.**

# What is JSON Web Token?

- **Compact**: Because of their smaller size, JWTs can be sent through a URL, POST parameter, or inside an HTTP header. Additionally, the smaller size means transmission is fast.

  Ex:
  eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJzdWIiOiIxMjM0NTY3ODkwIiwibmFtZSI6Ikpva
  G4gRG9lIiwiYWRtaW4iOnRydWV9.TJVA95OrM7E2cBab30RMHrHDcEfxjoYZgeFONFh7HgQ

- **Self-contained**: The payload contains all the required information about the user, avoiding the need to query the database more than once.

# When should you use JSON Web Tokens?

- **Authentication**: This is the most common scenario for using JWT. Once the user is logged in, each subsequent request will include the JWT, allowing the user to access routes, services, and resources that are permitted with that token. **Single Sign On** is a feature that widely uses JWT nowadays, because of its small overhead and its ability to be easily used across different domains.

- **Information Exchange**: JSON Web Tokens are a good way of securely transmitting information between parties. Because JWTs can be signed—for example, using public/private key pairs—you can be sure the senders are who they say they are.

# What is the JSON Web Token structure?

- JSON Web Tokens consist of three parts separated by dots (.), which are:
  - Header
  - Payload
  - Signature

Therefore, a JWT typically looks like the following.

- xxxxx.yyyyy.zzzzz

# What is the JSON Web Token structure?

## Header

The header *typically* consists of two parts: the type of the token, which is JWT, and the hashing algorithm being used, such as HMAC SHA256 or RSA.

- For example:

```
{
  "alg": "HS256",
  "typ": "JWT"
}
```

- Then, this JSON is **Base64Url** encoded to form the first part of the JWT.

# What is the JSON Web Token structure?

## Payload

The second part of the token is the payload, which contains the claims.

**Claims** are statements about an entity (typically, the user) and additional metadata. There are three types of claims:

- *reserved*
- *public*
- *private*

# What is the JSON Web Token structure?

## Payload

- **Reserved claims**

  These are a set of predefined claims which are not mandatory but recommended, to provide a set of useful, interoperable claims. Some of them are: **iss** (issuer), **exp** (expiration time), **sub**(subject), **aud** (audience), and others.

Notice that the claim names are only three characters long as JWT is meant to be compact.

# What is the JSON Web Token structure?

## Payload

- **Public claims**

  These can be defined at will by those using JWTs. But to avoid collisions they should be defined in the IANA JSON Web Token Registry or be defined as a URI that contains a collision resistant namespace.

# What is the JSON Web Token structure?

## Payload

- **Private claims**

  These are the custom claims created to share information between parties that agree on using them.

# What is the JSON Web Token structure?

## Payload

- **Example**

```
{
  "sub": "1234567890",
  "name": "John Doe",
  "admin": true
}
```

The payload is then **Base64Url** encoded to form the second part of the JSON Web Token.

# What is the JSON Web Token structure?

## Signature

To create the signature part you have to take the encoded header, the encoded payload, a secret, the algorithm specified in the header, and sign that.

- The signature is used to verify that the sender of the JWT is who it says it is and to ensure that the message wasn't changed along the way.
- For example if you want to use the HMAC SHA256 algorithm, the signature will be created in the following way:

```
HMACSHA256(
  base64UrlEncode(header) + "." +
  base64UrlEncode(payload),
  secret)
```

# What is the JSON Web Token structure?

**Putting all together**

The output is three Base64 strings separated by dots that can be easily passed in HTML and HTTP environments, while being more compact when compared to XML-based standards such as SAML.

The following shows a JWT that has the previous header and payload encoded, and it is signed with a secret.

```
eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.
eyJzdWIiOiIxMjM0NTY3ODkwIiwibmFtZSI6IkpvaG4
gRG9lIiwiaXNTb2NpYWwiOnRydWV9.
4pcPyMD09olPSyXnrXCjTwXyr4BsezdI1AVTmud2fU4
```

# How to test and see my JWT?

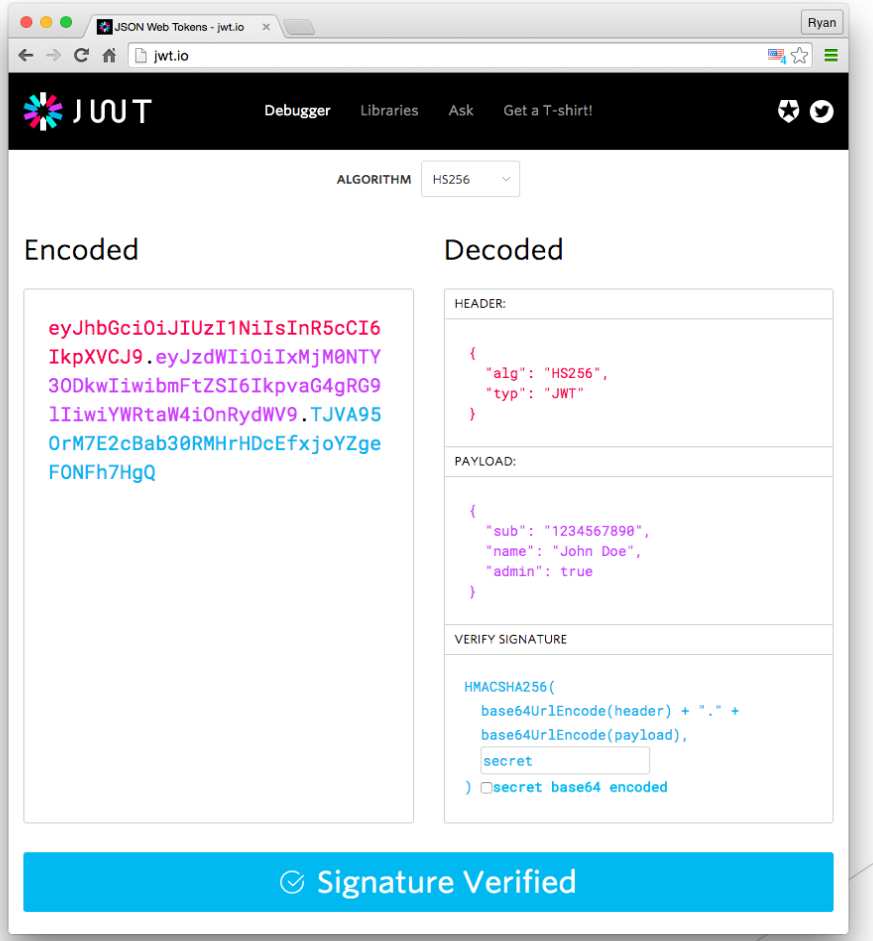
- **Jwt.io**

  It is a web page where you can learn more about JWT and debug a token.

  You can also verify the signature.

  And download the libraries for different languages as:

  Java, JS, Node.js, Python, .NET, etc.



# How do JSON Web Tokens work?

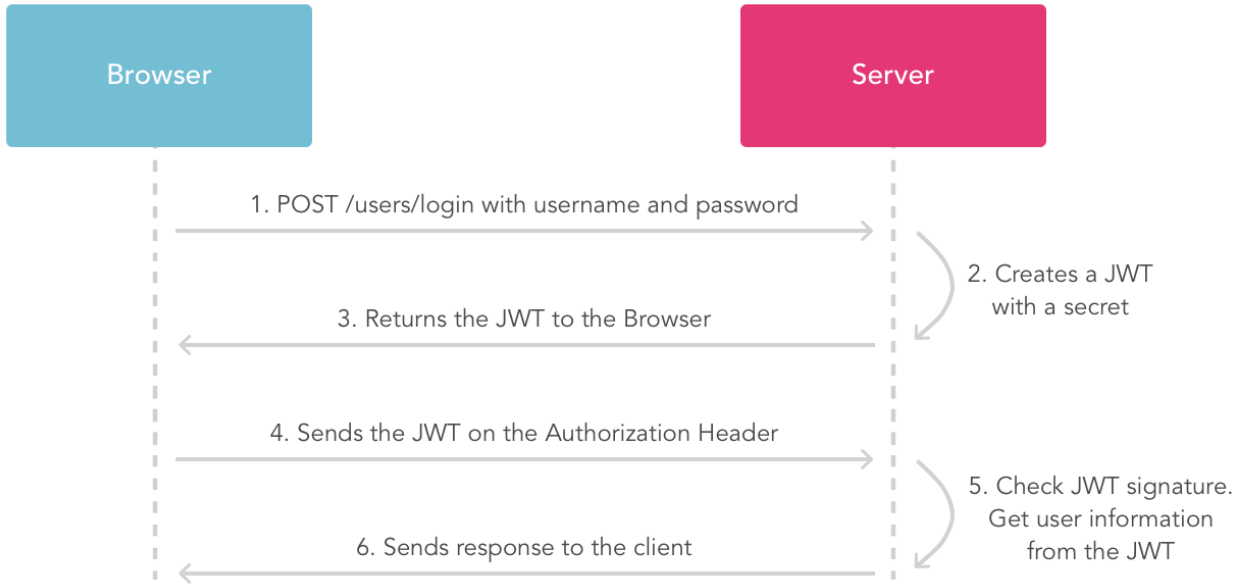
- In authentication, when the user successfully logs in using their credentials, a JSON Web Token will be returned and must be saved locally (typically in local storage, but cookies can be also used).
- Whenever the user wants to access a protected route or resource, the user agent should send the JWT, typically in the **Authorization** header using the **Bearer** schema. The content of the header should look like the following:

```
Authorization: Bearer <token>
```

- This is a stateless authentication mechanism as the user state is never saved in server memory. The server's protected routes will check for a valid JWT in the Authorization header, and if it's present, the user will be allowed to access protected resources. As JWTs are self-contained, all the necessary information is there, reducing the need to query the database multiple times.

# How do JSON Web Tokens work?

- This allows you to fully rely on data APIs that are stateless and even make requests to downstream services. It doesn't matter which domains are serving your APIs, so Cross-Origin Resource Sharing (CORS) won't be an issue as it doesn't use cookies.



# JWT Signature and Encryption

- A JWT is usually complemented with a signature or encryption. These are handled in their own specs as JSON Web Signature (JWS) and JSON Web Encryption (JWE).

- A signature allows a JWT to be *validated* against modifications. Encryption, on the other hand, makes sure the content of the JWT is only readable by certain parties.

# Common JWT Signing Algorithms

- Most JWTs in the wild are just signed. The most common algorithms are:
  - HMAC + SHA256
  - RSASSA-PKCS1-v1_5 + SHA256
  - ECDSA + P-256 + SHA256

The specs defines many more algorithms for signing. You can find them all in RFC 7518.

# Common JWT Signing Algorithms

**HMAC algorithms**

This is probably the most common algorithm for signed JWTs.

- Hash-Based Message Authentication Codes (HMACs) are a group of algorithms that provide a way of signing messages by means of a shared key. In the case of HMACs, a cryptographic hash function is used (for instance SHA256).

- The strength (i.e. how hard it is to forge an HMAC) depends on the hashing algorithm being used.

- The main objective in the design of the algorithm was to allow the combination of a key with a message while providing strong guarantees against tampering.

# Common JWT Signing Algorithms

**HMAC algorithms**

- HMACs are used with JWTs when you want a simple way for all parties to create and validate JWTs. Any party knowing the key can create new JWTs. In other words, with shared keys, it is possible for party to impersonate another one: HMAC JWTs do not provide guarantees with regards to the creator of the JWT. Anyone knowing the key can create one.

- For certain use cases, this is too permissive. This is where asymmetric algorithms come into play.

# Common JWT Signing Algorithms

RSA and ECDSA algorithms

- Both RSA and ECDSA are asymmetric encryption and digital signature algorithms.
- What asymmetric algorithms bring to the table is the possibility of verifying or decrypting a message without being able to create a new one.
- **This is key for certain use cases.**

# Common JWT Signing Algorithms

RSA and ECDSA algorithms

- Example: Picture a big company where data generated by the sales team needs to be verified by the accounting team.
  - If an HMAC were to be used to sign the data, then both the sales team and the accounting team would need to know the same key.
  - This would allow the sales team to sign data and make it pass as if it were from the accounting team.

- Although this might seem unlikely, especially in the context of a corporation, there are times when the ability to verify the creator of a signature is essential.

# Common JWT Signing Algorithms

RSA and ECDSA algorithms

- The main difference between RSA and ECDSA lies in speed and key size.
  - ECDSA requires smaller keys to achieve the same level of security as RSA. This makes it a great choice for small JWTs once is faster generating keys and signatures..
  - RSA, however, is usually faster than ECDSA for signature verification.
- As usual, pick the one that best aligns with your requirements.

# Conclusion

JWTs are a convenient way of representing authentication and authorization claims for your application.

- They are easy to parse, human readable and compact. But the killer features are in the JWS and JWE specs.
- With JWS and JWE all claims can be conveniently signed and encrypted, while remaining compact enough to be part of every API call

- Solutions such as session-ids and server-side tokens seem old and cumbersome when compared to the power of JWTs.

# Spring Security

# What is the Spring Security ?

- Spring Security is a framework that focuses on providing both authentication and authorization to Java applications.
- Like all Spring projects, the real power of Spring Security is found in how easily it can be extended to meet custom requirements.
- Features:
  - Comprehensive and extensible support for both Authentication and Authorization
  - Protection against attacks like session fixation, clickjacking, cross site request forgery, etc.
  - Servlet API integration
  - Optional integration with Spring Web MVC
  - Much more…

# Fundamentals

- Principal
  - User that performs the action
- Authentication
  - Confirming truth of credentials
- Authorization
  - Define access policy for principal
- GrantedAuthority
  - Application-wide permissions granted to a principal
- SecurityContext
  - Hold the Authentication and other security information
- SecurityContextHolder
  - Provide access to SecurityContext

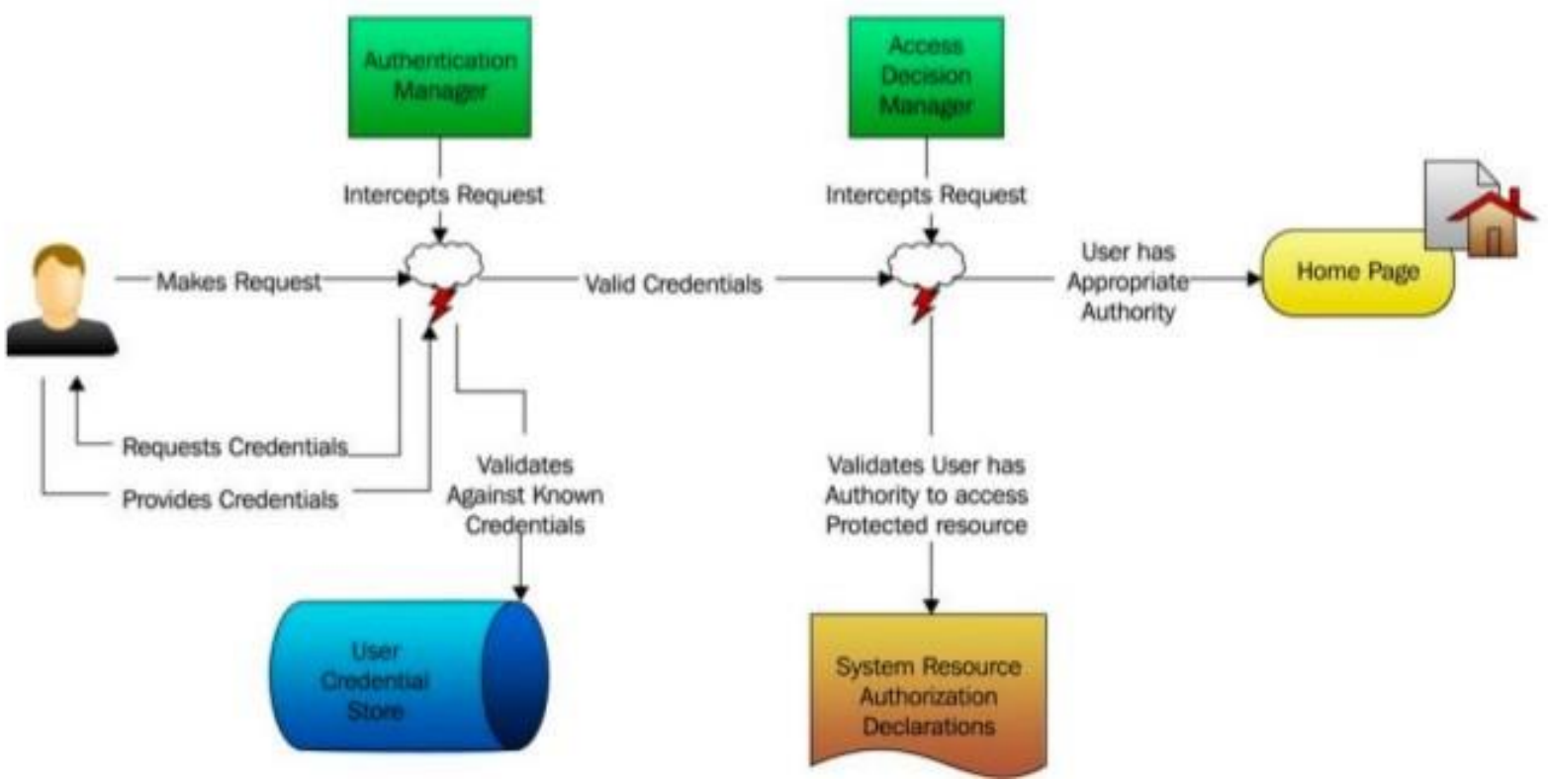
# SecurityContextHolder

- Provide access to SecurityContext
- Strategies
  - ThreadLocal – only read/write in the same thread
  - Global

```
SecurityContext context = SecurityContextHolder.getContext();
Object principal = context.getAuthentication().getPrincipal();

if (principal instanceof UserDetails) {
  String username = ((UserDetails)principal).getUsername();
} else {
  String username = principal.toString();
}
```

# Use Case


Authentication Manager
Access Decision Manager
Intercepts Request
Intercepts Request
Makes Request
Valid Credentials
User has Appropriate Authority
Home Page
Requests Credentials
Provides Credentials
Validates Against Known Credentials
Validates User has Authority to access Protected resource
User Credential Store
System Resource Authorization Declarations

# Basic filters

| Filter | Description |
| --- | --- |
| ChannelProcessingFilter | ensures that a request is being sent over HTTP or HTTPS |
| SecurityContextPersistentFilter | Populates the security context using information obtained from the repository (http session) |
| LogoutFilter | Used to log a user out of the application |
| UsernamePasswordAuthenticationFilter | Accepts the user's principal and credentials and attempts to authenticate the user |
| BasicAuthenticationFilter | Attempts to authenticate a user by processing an HTTP Basic authentication |
| ExceptionTranslationFilter | Handles any AccessDeniedException or AuthenticationException |
| FilterSecurityInterceptor | Decides whether or not to allow access to a secured resource |

http://static.springsource.org/spring-security/site/docs/3.0.x/reference/ns-config.html#ns-custom-filters

# Authentication

- Variants
  - Credential-based
  - Two-factor or 2FA
  - Hardware
- Mechanisms
  - Basic
  - Form
- Storage
  - RDBMS (Relational database managementsystem)
  - LDAP
  - Custom Storage

# Core Authentication service

- AuthenticationManager
  - Handles authentication requests
- AuthenticationProvider
  - Performs authentication
- UserDetailsService
  - Responsible for returning an UserDetails object
- UserDetails
  - Provides the core user information

# AuthenticationManager

```
public interface AuthenticationManager {
  /* Attempts to authenticate the passed Authentication object,
   * returning a fully populated Authentication object (including
   * granted authorities) if successful.
   * @param authentication the authentication request object
   * @return a fully authenticated object including credentials
   * @throws AuthenticationException if authentication fails */
  Authentication authenticate(Authentication authentication)
      throws AuthenticationException;
}
```

# AuthenticationProvider

```
public interface AuthenticationProvider {
  /* Performs authentication.
   * @param authentication the authentication request object.
   * @return a fully authenticated object including credentials.
   * @throws AuthenticationException if authentication fails.*/
  Authentication authenticate(Authentication authentication)
        throws AuthenticationException;

  /*Returns true if this provider supports the indicated
   *Authentication object.*/
  boolean supports(Class<? extends Object> authentication);
}
```

# UserDetailsService

```
/*Core interface which loads user-specific data.*/
public interface UserDetailsService {
  /* Locates the user based on the username.
   * @param username the username identifying the user
   * @return a fully populated user record (never null)
   * @throws UsernameNotFoundException if the user could not be
     found or the user has no GrantedAuthority
   * @throws DataAccessException if user could not be found for a
     repository-specific reason*/
  UserDetails loadUserByUsername(String username)
      throws UsernameNotFoundException, DataAccessException;
}
```

# UserDetails

```
/* Provides core user information.*/
public interface UserDetails extends Serializable {

  Collection<GrantedAuthority> getAuthorities();
  String getPassword();
  String getUsername();

  boolean isAccountNonExpired();
  boolean isAccountNonLocked();
  boolean isCredentialsNonExpired();
  boolean isEnabled();
}
```

# How to configure the Spring Security?

- The first step is to secure some routes of our application.
- For this demo we will expose the routes:
  - / and /login -> to everyone
  - /users -> to people whom can provide a valid JWT token.

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

<dependency>
    <groupId>io.jsonwebtoken</groupId>
    <artifactId>jjwt</artifactId>
    <version>0.7.0</version>
</dependency>
```

Once we have updated the pom.xml file and imported the new dependencies, we are ready to start securing our routes.

Ex: Maven Configuration

# How to configure the Spring Security?

- First of all, we want to avoid exposing /users to everyone, so we will create a configuration that restricts its access.
- We will accomplish this by adding a new class called WebSecurityConfig that extends the **WebSecurityConfigurerAdapter** class from Spring Security.

# How to configure the Spring Security?

```
@Configuration
@EnableWebSecurity
public class WebSecurityConfig extends WebSecurityConfigurerAdapter {
  @Override
  protected void configure(HttpSecurity http) throws Exception {
    http.csrf().disable().authorizeRequests()
        .antMatchers("/").permitAll()
        .antMatchers(HttpMethod.POST, "/login").permitAll()
        .anyRequest().authenticated()
        .and()
        // We filter the api/login requests
        .addFilterBefore(new JWTLoginFilter("/login", authenticationManager())
                UsernamePasswordAuthenticationFilter.class)
        // And filter other requests to check the presence of JWT in header
        .addFilterBefore(new JWTAuthenticationFilter(),
                UsernamePasswordAuthenticationFilter.class);
  }
```

- Here, we are specifying that / and /login are permitAll().
- All other requests are authenticated and:
- We are filtering login to add before the filter of users
- Any other endpoint, check the present of the JWT Token

# How to configure the Spring Security?

- We also configure from WHERE we are getting the users, where are 2 options:
  - inMemoryAuthentication() – Username and password pre-defined (good for tests).
  - userDetailsService() – You can declare a Service class to authenticate/authorize. Needs to implement **UserDetailsService** interface.

```
@Override
protected void configure(AuthenticationManagerBuilder auth) throws Exception {
  auth.userDetailsService(customMongoSecurityService).and()
          .inMemoryAuthentication()
          .withUser( username: "user").password( password: "user").roles(ROLE_USER).and()
          .withUser( username: "admin").password( password: "1234").roles(ROLE_ADMIN)
}
```

# Custom UserService

```groovy
/**
 * Custom service to retrieve users from database (@see UserDetailsService).
 * This is looking for a user in a MongoDB and converting to a org.springframework.security.core.userdetails.User
 */
@Service
class CustomMongoSecurityService implements UserDetailsService {

    @Autowired
    UserRepository userRepository

    @Override
    UserDetails loadUserByUsername(String username)
            throws UsernameNotFoundException {

        User user = userRepository.findByUsername(username)

        if (user) {
            return new org.springframework.security.core.userdetails.User(
                    user.getUsername(), user.getPassword(), getAuthorities(user))
        }

        throw new UsernameNotFoundException("No user found for $username")
    }

    private static Collection<? extends GrantedAuthority> getAuthorities(User user) {
        List authorities = new ArrayList()
        user.roles.each { authorities.add(new SimpleGrantedAuthority(it.getRole())) }
        authorities
    }
}
```

```groovy
@ToString
@Document(collection="user")
class User {
  @Id
  String id
  String username
  String password
  Set<Role> roles
}
```

# What about securing REST applications?

- The previous examples were normally for web applications, where you redirect pages, login using page, etc. In REST, we don't have:
  - Login page
  - Page to redirect after login
  - Page to redirect in failure or unauthorized

- Solution:
  - Override **AuthenticationFailureHandler** to return 401
  - Override **AuthenticationSuccessHandler** to return the JSON object / token.
  - Override **AuthenticationEntryPoint** to always return 401.
  - Override **LogoutSuccessHandler** to return 200.

# Overriding the **AuthenticationEntryPoint**

- Class extends org.springframework.security.web.AuthenticationEntryPoint, and implements only one method, which sends response error (with 401 status code) in cause of unauthorized attempt.

```
@Component
public class HttpAuthenticationEntryPoint implements AuthenticationEntryPoint {
    @Override
    public void commence(HttpServletRequest request, HttpServletResponse response,
            AuthenticationException authException) throws IOException {
        response.sendError(HttpServletResponse.SC_UNAUTHORIZED, authException.getMessage());
    }
}
```

# Overriding the AuthenticationSuccessHandler

- The AuthenticationSuccessHandler is responsible of what to do after a successful authentication, by default it will redirect to an URL, but in our case we want it to send an HTTP response with data.

```
@Component
public class AuthSuccessHandler extends SavedRequestAwareAuthenticationSuccessHandler {
    private static final Logger LOGGER = LoggerFactory.getLogger(AuthSuccessHandler.class);

    private final ObjectMapper mapper;

    @Autowired
    AuthSuccessHandler(MappingJackson2HttpMessageConverter messageConverter) {
        this.mapper = messageConverter.getObjectMapper();
    }

    @Override
    public void onAuthenticationSuccess(HttpServletRequest request, HttpServletResponse response,
            Authentication authentication) throws IOException, ServletException {
        response.setStatus(HttpServletResponse.SC_OK);

        NuvolaUserDetails userDetails = (NuvolaUserDetails) authentication.getPrincipal();
        User user = userDetails.getUser();
        userDetails.setUser(user);

        LOGGER.info(userDetails.getUsername() + " got is connected ");

        PrintWriter writer = response.getWriter();
        mapper.writeValue(writer, user);
        writer.flush();
    }
}
```

# Overriding the **AuthenticationFailureHandler**

- The AuthenticationFaillureHandler is responsible of what to after a failed authentication, by default it will redirect to the login page URL, but in our case we just want it to send an HTTP response with the 401 UNAUTHORIZED code.

```
@Component
public class AuthFailureHandler extends SimpleUrlAuthenticationFailureHandler {
    @Override
    public void onAuthenticationFailure(HttpServletRequest request, HttpServletResponse response,
            AuthenticationException exception) throws IOException, ServletException {
        response.setStatus(HttpServletResponse.SC_UNAUTHORIZED);

        PrintWriter writer = response.getWriter();
        writer.write(exception.getMessage());
        writer.flush();
    }
}
```