**Spring cloud Security With Oauth2**

Spring Security is a framework which provides various security features like: authentication, authorization to create secure Java Enterprise Applications.

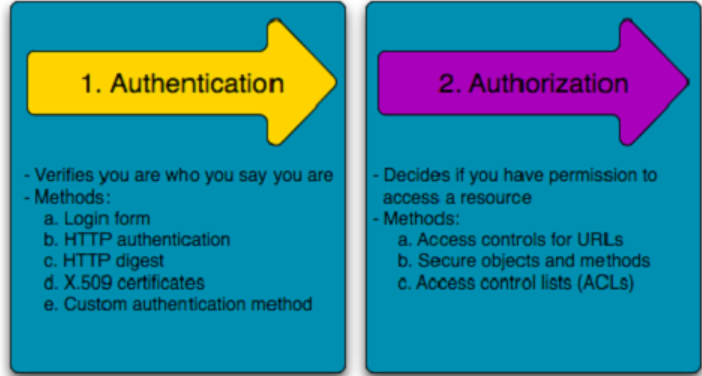
### What is Authentication?

Authentication refers to the validation of credentials such as user ID/ user name / password, etc. in order to verify the identity of any given user.

### What is Authorization?

Authorization takes place once the user identity has been validated and successfully authenticated by any given system. Authorization allows a user to use system resources of the likes of information, databases, funds, locations, permission.

We can apply authorization to authorize web request, methods and access to individual domain.



**Default Security Setup**

Spring Boot application, we need to add the *security starter dependency*:

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

Step 1**: Create class as SecurityConfig class add annotation EnableWebSecurity and extends abstract class of WebSecurityConfigurerAdapter and overload configures methods**.

In Memory configuration

|  |
| --- |
| **import** org.springframework.context.annotation.Configuration;  **import** org.springframework.security.config.annotation.authentication.builders.AuthenticationManagerBuilder;  **import** org.springframework.security.config.annotation.web.builders.HttpSecurity;  **import** org.springframework.security.config.annotation.web.configuration.EnableWebSecurity;  **import** org.springframework.security.config.annotation.web.configuration.WebSecurityConfigurerAdapter;  **import** org.springframework.security.crypto.factory.PasswordEncoderFactories;  **import** org.springframework.security.crypto.password.PasswordEncoder;  @Configuration  @EnableWebSecurity  **public** **class** SecurityConfig **extends** WebSecurityConfigurerAdapter {  @Override  **protected** **void** configure(AuthenticationManagerBuilder auth) **throws** Exception {  PasswordEncoder encoder = PasswordEncoderFactories.*createDelegatingPasswordEncoder*();  auth.inMemoryAuthentication().withUser("nag").password(encoder.encode("nag")).roles("user");  auth.inMemoryAuthentication().withUser("admin").password(encoder.encode("admin")).roles("user","admin");  }  @Override  **protected** **void** configure(HttpSecurity http) **throws** Exception {  http.csrf().disable().authorizeRequests().antMatchers("/product/user/\*\*").hasAnyRole("user", "admin").and().formLogin();  http.csrf().disable().authorizeRequests().antMatchers("/product/admin/\*\*").hasAnyRole("admin").and().formLogin();  }  }  With chaining:  PasswordEncoder encoder = PasswordEncoderFactories.*createDelegatingPasswordEncoder*();  auth.inMemoryAuthentication().withUser("nag").password(encoder.encode("nag")).roles("user") .and().withUser("admin").password(encoder.encode("admin")).roles("user", "admin");  http.csrf().disable().authorizeRequests().antMatchers("/product/user/\*\*").hasAnyRole("user", "admin").and().authorizeRequests().antMatchers("/product/admin/\*\*").hasAnyRole("admin").and().formLogin();  To discard the security auto-configuration and add our own configuration, we need to exclude the SecurityAutoConfiguration class.  @SpringBootApplication(exclude = { SecurityAutoConfiguration.class })  public class SpringBootSecurityApplication {    public static void main(String[] args) {  SpringApplication.run(SpringBootSecurityApplication.class, args);  }  } |

|  |
| --- |
| @Configuration  @EnableWebSecurity  @EnableGlobalMethodSecurity(prePostEnabled=**true**)  **public** **class** SecurityConfig **extends** WebSecurityConfigurerAdapter {  @Autowired  **private** UserDetailsService userDetailsService;  @Override  **protected** **void** configure(AuthenticationManagerBuilder auth) **throws** Exception {  auth.userDetailsService(userDetailsService).passwordEncoder(encoderPassword());  }  @Override  **protected** **void** configure(HttpSecurity http) **throws** Exception {  http.csrf().disable();  http.authorizeRequests().antMatchers("/rest/\*\*").authenticated().anyRequest().permitAll().and()  .authorizeRequests().antMatchers("/secure/\*\*").authenticated().anyRequest().hasAnyRole("ADMIN").and()  .formLogin()  .permitAll();  }  @Bean(name = "passwordEncoder")  **public** BCryptPasswordEncoder encoderPassword() {  **return** **new** BCryptPasswordEncoder();  }  @Entity  **public** **class** Role {  @Id  @GeneratedValue  **private** **int** role\_id;  **private** String role;    **public** Role() {}  **public** Role(**int** role\_id, String role) {  **this**.role\_id = role\_id;  **this**.role = role;  }  **public** **int** getRole\_id() {  **return** role\_id;  }  **public** **void** setRole\_id(**int** role\_id) {  **this**.role\_id = role\_id;  }  **public** String getRole() {  **return** role;  }  **public** **void** setRole(String role) {  **this**.role = role;  }    }  }  @Entity  **public** **class** User {    @Id  **private** **int** user\_id;  **private** String username;  **private** String password;  **private** String email;    @OneToMany(cascade=CascadeType.***ALL***, fetch=FetchType.***EAGER***)  @JoinTable(name="user\_role", joinColumns=@JoinColumn(name="user\_id"), inverseJoinColumns=@JoinColumn(name="role\_id"))  **private** Set<Role> roles;    **public** User() {}  **public** User(**int** user\_id, String username, String password, String email, Set<Role> roles) {  **this**.user\_id = user\_id;  **this**.username = username;  **this**.password = password;  **this**.email = email;  **this**.roles = roles;  }  **public** **int** getUser\_id() {  **return** user\_id;  }  **public** **void** setUser\_id(**int** user\_id) {  **this**.user\_id = user\_id;  }  **public** String getUsername() {  **return** username;  }  **public** **void** setUsername(String username) {  **this**.username = username;  }  **public** String getPassword() {  **return** password;  }  **public** **void** setPassword(String password) {  **this**.password = password;  }  **public** String getEmail() {  **return** email;  }  **public** **void** setEmail(String email) {  **this**.email = email;  }  **public** Set<Role> getRoles() {  **return** roles;  }  **public** **void** setRoles(Set<Role> roles) {  **this**.roles = roles;  }  }  UserDetailsService interface is used to retrieve user-related data. It has one method named loadUserByUsername() which can be overridden to customize the process of finding the user.  **import** javax.persistence.EntityManager;  **import** javax.persistence.PersistenceContext;  **import** org.springframework.security.core.userdetails.UserDetails;  **import** org.springframework.security.core.userdetails.UserDetailsService;  **import** org.springframework.security.core.userdetails.UsernameNotFoundException;  **import** org.springframework.stereotype.Service;  **import** com.mng.java.model.User;  @Service  **public** **class** MyUserDetailsServiceImp **implements** UserDetailsService {    @PersistenceContext  **private** EntityManager entityManager;  @Override  **public** UserDetails loadUserByUsername(String username) {    User user = **new** User();  @SuppressWarnings("unchecked")  List<User> list = entityManager.createQuery("SELECT u FROM User u WHERE u.username = :username")  .setParameter("username", username)  .getResultList();    **if** (!list.isEmpty()) {  user = (User) list.get(0);  System.***out***.println("\*\*\*User Object\*\*\*: "+ user.getUsername());  }**else** {  **throw** **new** UsernameNotFoundException("user not exist with name: "+username);  }    MyUserDetailsImpl detailsImpl = **new** MyUserDetailsImpl();  detailsImpl.setUser(user);  **return** detailsImpl;  }  }  **UserDetails Interface implemention**  **import** org.springframework.security.core.GrantedAuthority;  **import** org.springframework.security.core.authority.SimpleGrantedAuthority;  **import** org.springframework.security.core.userdetails.UserDetails;  **import** com.mng.java.model.User;  **public** **class** MyUserDetailsImpl **implements** UserDetails {  **private** **static** **final** **long** ***serialVersionUID*** = 1L;  **private** User user;  **public** User getUser() {  **return** user;  }  **public** **void** setUser(User user) {  **this**.user = user;  }  @Override  **public** Collection<? **extends** GrantedAuthority> getAuthorities() {  **return** user.getRoles().stream().map(role -> **new** SimpleGrantedAuthority("ROLE\_"+ role)).collect(Collectors.*toList*());  }  @Override  **public** String getPassword() {  **return** user.getPassword();  }  @Override  **public** String getUsername() {  **return** user.getUsername();  }  @Override  **public** **boolean** isAccountNonExpired() {  **return** **true**;  }  @Override  **public** **boolean** isAccountNonLocked() {  **return** **true**;  }  @Override  **public** **boolean** isCredentialsNonExpired() {  **return** **true**;  }  @Override  **public** **boolean** isEnabled() {  **return** **true**;  }  }  @Repository  @Transactional  **public** **class** UserDaoImpl **implements** UserDao {    @PersistenceContext  **private** EntityManager entityManager;  @Override  **public** **void** save(User user) {  entityManager.persist(user);  }  }  **public** **interface** UserDao {  **public** **void** save(User use);  } |

Database Authentication

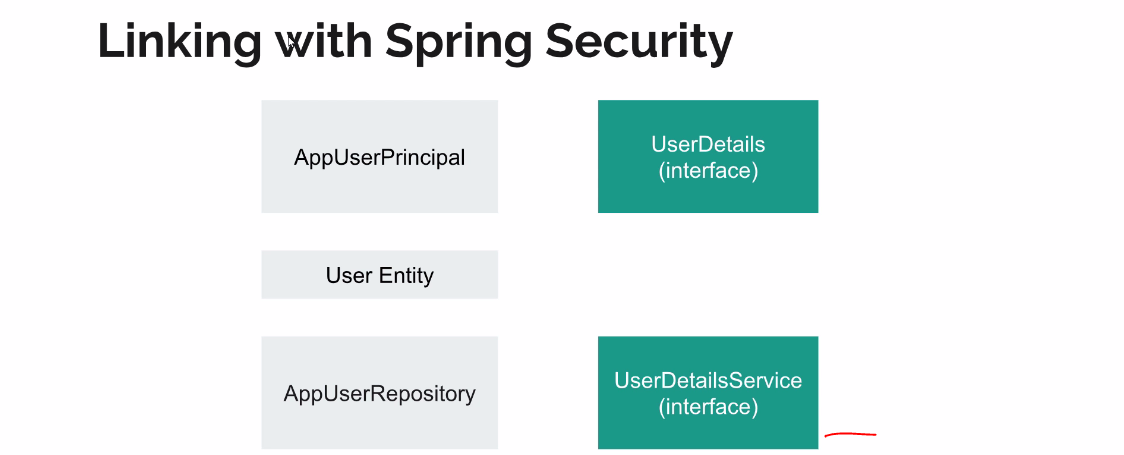
Step 1: Create user entity to store user information

Step 2: Store the user in our database. JPARepository.

Step 3: Link our User entity with the built in classes in spring security.

* Link User with UserDetails interface
* Link UserRepository with UserDetailsService interface.

Step 4: integrate database Auth in out Configuration



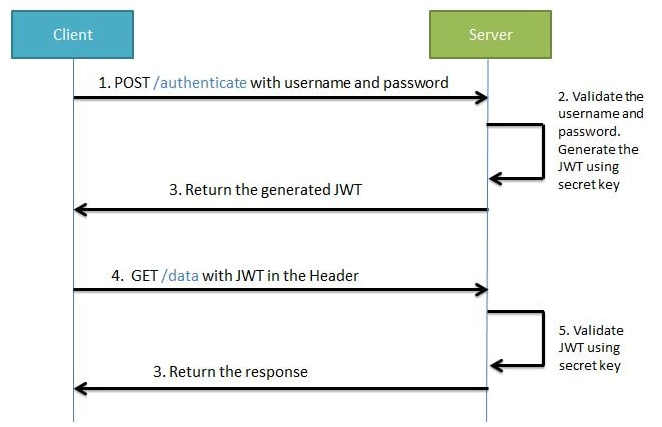
**JWT Web Token**

A JSON Web Token (JWT) is an open standard ([RFC 7519](https://tools.ietf.org/html/rfc7519)) 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 or a public/private key pair

The client will need to authenticate with the server using the credentials only once. During this time the server validates the credentials and returns the client a JSON Web Token(JWT). For all future requests the client can authenticate itself to the server using this JSON Web Token(JWT) and so does not need to send the credentials like username and password.

**Workflow of how JWT is used**

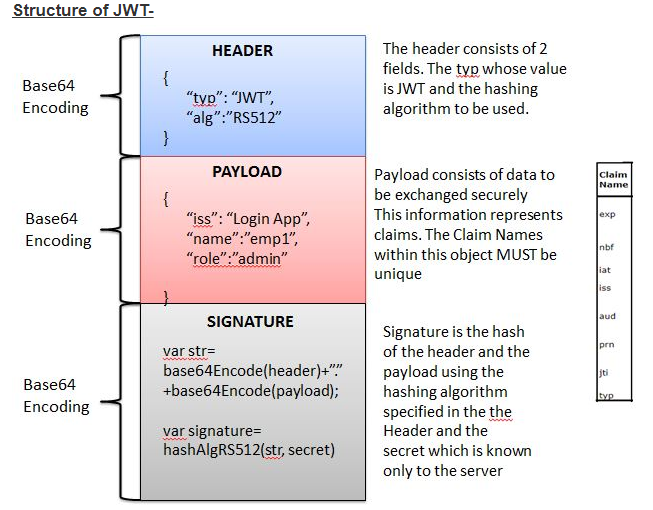
[**https://www.javainuse.com/spring/jwt**](https://www.javainuse.com/spring/jwt)



During the first request the client sends a POST request with username and password. Upon successful authentication the server generates the JWT sends this JWT to the client. This JWT can contain a payload of data. On all subsequent requests the client sends this JWT token in the header. Using this token the server authenticates the user. So we don't need the client to send the user name and password to the server during each request for authentication, but only once after which the server issues a JWT to the client. A JWT payload can contain things like user ID so that when the client again sends the JWT, you can be sure that it is issued by you, and you can see to whom it was issued.

Structure of JWT

JWT has the following format -**header.payload.signature**   

An important point to remember about JWT is that the information in the payload of the JWT is visible to everyone. So we should not pass any sensitive information like passwords in the payload. We can encrypt the payload data if we want to make it more secure. However we can be sure that no one can tamper and change the payload information. If this is done the server will recognize it

# Step 1. Create the ****HEADER****

The header component of the JWT contains information about how the JWT signature should be computed. The header is a JSON object in the following format:

In this JSON, the value of the “typ” key specifies that the object is a JWT, and the value of the “alg” key specifies which hashing algorithm is being used to create the JWT signature component. In our example, we’re using the HMAC-SHA256 algorithm, a hashing algorithm that uses a secret key, to compute the signature

# Step 2. Create the PAYLOAD

The payload component of the JWT is the data that‘s stored inside the JWT (this data is also referred to as the “claims” of the JWT). In our example, the authentication server creates a JWT with the user information stored inside of it, specifically the user ID.

The data inside the payload is referred to as the “claims” of the token.

In our example, we are only putting one claim into the payload. You can put as many claims as you like. There are several different standard claims for the JWT payload, such as “iss” the issuer, “sub” the subject, and “exp” the expiration time. These fields can be useful when creating JWT, but they are optional. See the [wikipedia page](https://en.wikipedia.org/wiki/JSON_Web_Token" \l "Standard_fields) on JWT for a more detailed list of JWT standard fields.

Keep in mind that the size of the data will affect the overall size of the JWT, this generally isn’t an issue but having excessively large JWT may negatively affect performance and cause latency.

# Step 3. Create the SIGNATURE

The signature is computed using the following pseudo code:

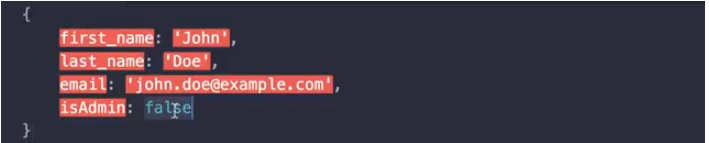
// signature algorithmdata = base64urlEncode( header ) + “.” + base64urlEncode( payload )

hashedData = hash( data, secret )

signature = base64urlEncode( hashedData )

What this algorithm does is [base64url encodes](http://kjur.github.io/jsjws/tool_b64uenc.html) the header and the payload created in steps 1 and 2. The algorithm then joins the resulting encoded strings together with a period (.) in between them. In our pseudo code, this joined string is assigned to data. The data string is [hashed](https://en.wikipedia.org/wiki/Hash_function) with the secret key using the hashing algorithm specified in the JWT header. The resulting hashed data is assigned to hashedData. This hashed data is then base64url encoded to produce the JWT signature.

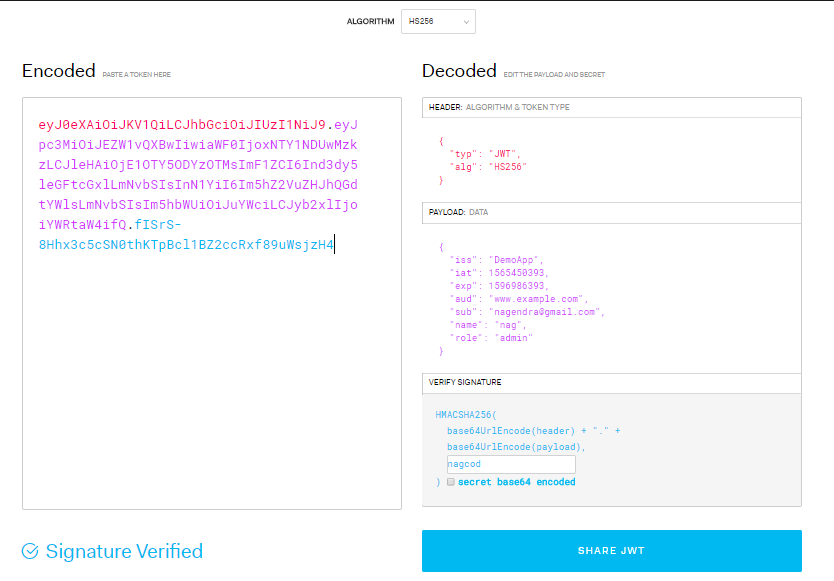
Claim is nothing but below like first name, last name, email and additionally expire date, once is token is generated, information signed ,we can’t possible to change this content.



<https://www.javainuse.com/spring/jwt>

**Inspect the contents of the created token**

We will be inspecting JWT token using [JWT Online Decoder](https://jwt.io/)



Token base64 encode we can easy to decode like

<https://medium.com/better-programming/secure-a-spring-boot-rest-api-with-json-web-token-reference-to-angular-integration-e57a25806c50>

<https://medium.com/omarelgabrys-blog/microservices-with-spring-boot-authentication-with-jwt-part-3-fafc9d7187e8>

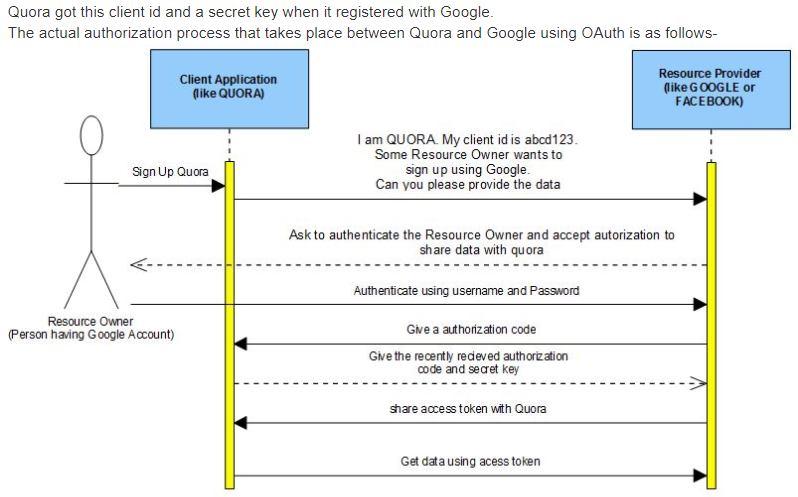
<https://dzone.com/articles/angular-7-spring-boot-jwt-authentication-example?fromrel=true>

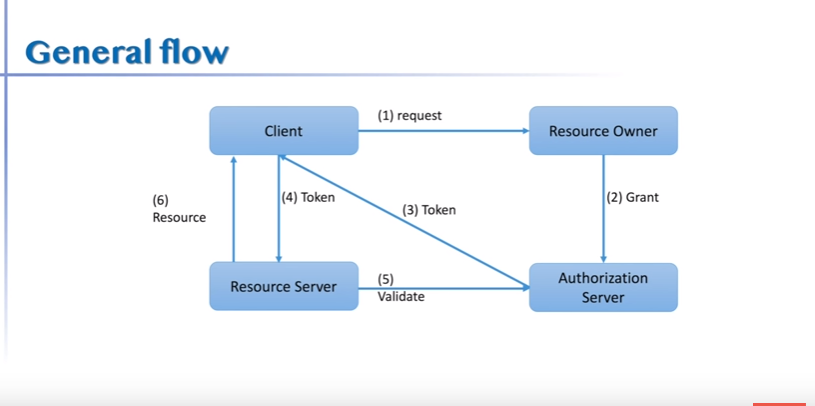
OAuth stands for Open Authorization. It’s a free and open protocol and OAuth2 is an authorization framework, it enable the application to get the limited access to user accounts on an http service such ad facebook, github, google, linkedlin.

It authorize the third party applications to access the user account.

In the above example of Quora, we have 3 actors-

* **Resource Owner** - This is the user who wants to sign up using Quora.
* **Client Application** - This will be Quora
* **Resource Server** - This will be Gmail or Facebook.
* **Authorization Server** - The resource server hosts the protected user accounts, and the authorization server verifies the identity of the user then issues access tokens to the application.





. 