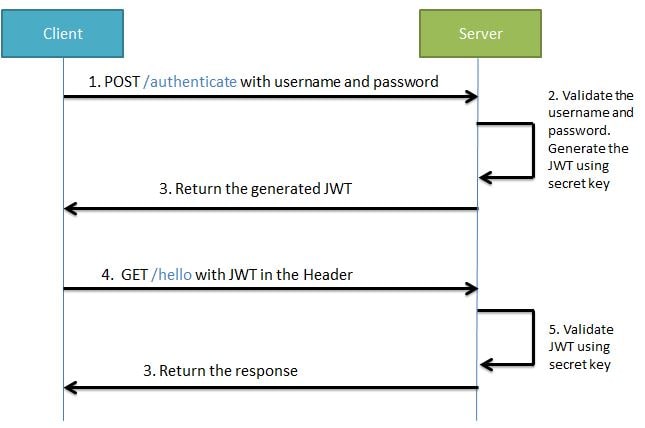
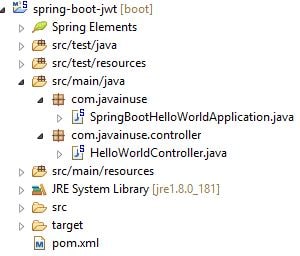
* Develop a Spring Boot Application to expose a Simple REST GET API with mapping /hello.
* Configure Spring Security for JWT. Expose REST POST API with mapping /authenticate using which User will get a valid JSON Web Token. And then allow the user access to the api /hello only if it has a valid token  
  

Develop a Spring Boot Application to expose a GET REST API

Maven Project will be as follows-  
  


The pom.xml is as follows-

<project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"

xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/maven-4.0.0.xsd">

<modelVersion>4.0.0</modelVersion>

<groupId>com.javainuse</groupId>

<artifactId>spring-boot-jwt</artifactId>

<version>0.0.1-SNAPSHOT</version>

<parent>

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

<artifactId>spring-boot-starter-parent</artifactId>

<version>2.1.1.RELEASE</version>

<relativePath /> <!-- lookup parent from repository -->

</parent>

<properties>

<project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>

<project.reporting.outputEncoding>UTF-8</project.reporting.outputEncoding>

<java.version>1.8</java.version>

</properties>

<dependencies>

<dependency>

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

<artifactId>spring-boot-starter-web</artifactId>

</dependency>

</dependencies>

</project>

Create a Controller class for exposing a GET REST API-

package com.javainuse.controller;

import org.springframework.web.bind.annotation.RequestMapping;

import org.springframework.web.bind.annotation.RestController;

@RestController

public class HelloWorldController {

@RequestMapping({ "/hello" })

public String firstPage() {

return "Hello World";

}

}

Create the bootstrap class with SpringBoot Annotation

package com.javainuse;

import org.springframework.boot.SpringApplication;

import org.springframework.boot.autoconfigure.SpringBootApplication;

@SpringBootApplication

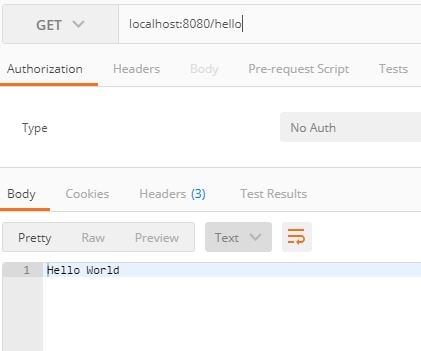
public class SpringBootHelloWorldApplication {

public static void main(String[] args) {

SpringApplication.run(SpringBootHelloWorldApplication.class, args);

}

}

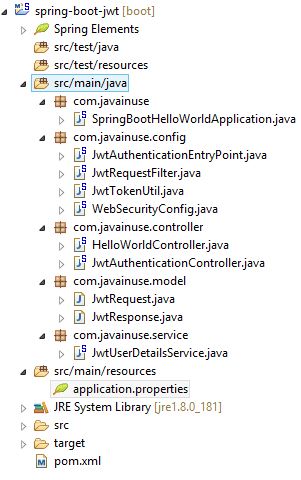
Compile and the run the SpringBootHelloWorldApplication.java as a Java application.  
Go to **localhost:8080/hello**  


Spring Security and JWT Configuration

We will be configuring Spring Security and JWT for performing 2 operations-

* **Generating JWT** - Expose a POST API with mapping **/authenticate**. On passing correct username and password it will generate a JSON Web Token(JWT)
* **Validating JWT** - If user tries to access GET API with mapping **/hello**. It will allow access only if request has a valid JSON Web Token(JWT)

Maven Project will be as follows-

  
The sequence flow for these operations will be as follows-

Generating JWT


Validating JWT

  
Add the Spring Security and JWT dependencies

<project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"

xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/maven-4.0.0.xsd">

<modelVersion>4.0.0</modelVersion>

<groupId>com.javainuse</groupId>

<artifactId>spring-boot-jwt</artifactId>

<version>0.0.1-SNAPSHOT</version>

<parent>

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

<artifactId>spring-boot-starter-parent</artifactId>

<version>2.1.1.RELEASE</version>

<relativePath /> <!-- lookup parent from repository -->

</parent>

<properties>

<project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>

<project.reporting.outputEncoding>UTF-8</project.reporting.outputEncoding>

<java.version>1.8</java.version>

</properties>

<dependencies>

<dependency>

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

<artifactId>spring-boot-starter-web</artifactId>

</dependency>

**<dependency>**

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

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

**</dependency>**

**<dependency>**

**<groupId>io.jsonwebtoken</groupId>**

**<artifactId>jjwt</artifactId>**

**<version>0.9.1</version>**

**</dependency>**

</dependencies>

</project>

* Define the application.properties. As see in [previous JWT tutorial, we specify the secret key using which we will be using for hashing algorithm.](https://www.javainuse.com/spring/jwt) The secret key is combined with the header and the payload to create a unique hash. We are only able to verify this hash if you have the secret key.
* jwt.secret=javainuse
* JwtTokenUtil

The JwtTokenUtil is responsible for performing JWT operations like creation and validation.It makes use of the io.jsonwebtoken.Jwts for achieving this.

package com.javainuse.config;

import java.io.Serializable;

import java.util.Date;

import java.util.HashMap;

import java.util.Map;

import java.util.function.Function;

import org.springframework.beans.factory.annotation.Value;

import org.springframework.security.core.userdetails.UserDetails;

import org.springframework.stereotype.Component;

import io.jsonwebtoken.Claims;

import io.jsonwebtoken.Jwts;

import io.jsonwebtoken.SignatureAlgorithm;

@Component

public class JwtTokenUtil implements Serializable {

private static final long serialVersionUID = -2550185165626007488L;

public static final long JWT\_TOKEN\_VALIDITY = 5 \* 60 \* 60;

@Value("${jwt.secret}")

private String secret;

**//retrieve username from jwt token**

public String getUsernameFromToken(String token) {

return getClaimFromToken(token, Claims::getSubject);

}

**//retrieve expiration date from jwt token**

public Date getExpirationDateFromToken(String token) {

return getClaimFromToken(token, Claims::getExpiration);

}

public <T> T getClaimFromToken(String token, Function<Claims, T> claimsResolver) {

final Claims claims = getAllClaimsFromToken(token);

return claimsResolver.apply(claims);

}

**//for retrieveing any information from token we will need the secret key**

private Claims getAllClaimsFromToken(String token) {

return Jwts.parser().setSigningKey(secret).parseClaimsJws(token).getBody();

}

**//check if the token has expired**

private Boolean isTokenExpired(String token) {

final Date expiration = getExpirationDateFromToken(token);

return expiration.before(new Date());

}

**//generate token for user**

public String generateToken(UserDetails userDetails) {

Map<String, Object> claims = new HashMap<>();

return doGenerateToken(claims, userDetails.getUsername());

}

**//while creating the token -**

**//1. Define claims of the token, like Issuer, Expiration, Subject, and the ID**

**//2. Sign the JWT using the HS512 algorithm and secret key.**

**//3. According to JWS Compact Serialization(https://tools.ietf.org/html/draft-ietf-jose-json-web-signature-41#section-3.1)**

**// compaction of the JWT to a URL-safe string**

private String doGenerateToken(Map<String, Object> claims, String subject) {

return Jwts.builder().setClaims(claims).setSubject(subject).setIssuedAt(new Date(System.currentTimeMillis()))

.setExpiration(new Date(System.currentTimeMillis() + JWT\_TOKEN\_VALIDITY \* 1000))

.signWith(SignatureAlgorithm.HS512, secret).compact();

}

**//validate token**

public Boolean validateToken(String token, UserDetails userDetails) {

final String username = getUsernameFromToken(token);

return (username.equals(userDetails.getUsername()) && !isTokenExpired(token));

}

}

* JWTUserDetailsService

JWTUserDetailsService implements the Spring Security UserDetailsService interface. It overrides the loadUserByUsername for fetching user details from the database using the username. The Spring Security Authentication Manager calls this method for getting the user details from the database when authenticating the user details provided by the user. Here we are getting the **user details from a hardcoded User List**. In the [next tutorial we will be adding the DAO implementation for fetching User Details from the Database.](https://www.javainuse.com/spring/boot-jwt-mysql) Also the password for a user is stored in encrypted format using BCrypt. Previously we have seen [Spring Boot Security - Password Encoding Using Bcrypt.](https://www.javainuse.com/spring/boot_security_jdbc_authentication_bcrypt) Here using the [Online Bcrypt Generator you can generate the Bcrypt for a password.](https://www.javainuse.com/onlineBcrypt)

package com.javainuse.service;

import java.util.ArrayList;

import org.springframework.security.core.userdetails.User;

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;

@Service

public class JwtUserDetailsService implements UserDetailsService {

@Override

public UserDetails loadUserByUsername(String username) throws UsernameNotFoundException {

**if ("javainuse".equals(username)) {**

**return new User("javainuse", "$2a$10$slYQmyNdGzTn7ZLBXBChFOC9f6kFjAqPhccnP6DxlWXx2lPk1C3G6",**

**new ArrayList<>());**

**}** else {

throw new UsernameNotFoundException("User not found with username: " + username);

}

}

}

* JwtAuthenticationController

Expose a POST API /authenticate using the JwtAuthenticationController. The POST API gets username and password in the body- Using Spring Authentication Manager we authenticate the username and password.If the credentials are valid, a JWT token is created using the JWTTokenUtil and provided to the client.

package com.javainuse.controller;

import java.util.Objects;

import org.springframework.beans.factory.annotation.Autowired;

import org.springframework.http.ResponseEntity;

import org.springframework.security.authentication.AuthenticationManager;

import org.springframework.security.authentication.BadCredentialsException;

import org.springframework.security.authentication.DisabledException;

import org.springframework.security.authentication.UsernamePasswordAuthenticationToken;

import org.springframework.security.core.userdetails.UserDetails;

import org.springframework.web.bind.annotation.CrossOrigin;

import org.springframework.web.bind.annotation.RequestBody;

import org.springframework.web.bind.annotation.RequestMapping;

import org.springframework.web.bind.annotation.RequestMethod;

import org.springframework.web.bind.annotation.RestController;

import com.javainuse.service.JwtUserDetailsService;

import com.javainuse.config.JwtTokenUtil;

import com.javainuse.model.JwtRequest;

import com.javainuse.model.JwtResponse;

@RestController

@CrossOrigin

public class JwtAuthenticationController {

@Autowired

private AuthenticationManager authenticationManager;

@Autowired

private JwtTokenUtil jwtTokenUtil;

@Autowired

private JwtUserDetailsService userDetailsService;

@RequestMapping(value = "/authenticate", method = RequestMethod.POST)

public ResponseEntity<?> createAuthenticationToken(@RequestBody JwtRequest authenticationRequest) throws Exception {

authenticate(authenticationRequest.getUsername(), authenticationRequest.getPassword());

final UserDetails userDetails = userDetailsService

.loadUserByUsername(authenticationRequest.getUsername());

final String token = jwtTokenUtil.generateToken(userDetails);

return ResponseEntity.ok(new JwtResponse(token));

}

private void authenticate(String username, String password) throws Exception {

try {

**authenticationManager.authenticate(new UsernamePasswordAuthenticationToken(username, password));**

} catch (DisabledException e) {

throw new Exception("USER\_DISABLED", e);

} catch (BadCredentialsException e) {

throw new Exception("INVALID\_CREDENTIALS", e);

}

}

}

* JwtRequest

This class is required for storing the username and password we recieve from the client.

package com.javainuse.model;

import java.io.Serializable;

public class JwtRequest implements Serializable {

private static final long serialVersionUID = 5926468583005150707L;

private String username;

private String password;

//need default constructor for JSON Parsing

public JwtRequest()

{

}

public JwtRequest(String username, String password) {

this.setUsername(username);

this.setPassword(password);

}

public String getUsername() {

return this.username;

}

public void setUsername(String username) {

this.username = username;

}

public String getPassword() {

return this.password;

}

public void setPassword(String password) {

this.password = password;

}

}

* JwtResponse

This is class is required for creating a response containing the JWT to be returned to the user.

package com.javainuse.model;

import java.io.Serializable;

public class JwtResponse implements Serializable {

private static final long serialVersionUID = -8091879091924046844L;

private final String jwttoken;

public JwtResponse(String jwttoken) {

this.jwttoken = jwttoken;

}

public String getToken() {

return this.jwttoken;

}

}

* JwtRequestFilter

The JwtRequestFilter extends the Spring Web Filter OncePerRequestFilter class. For any incoming request this Filter class gets executed. It checks if the request has a valid JWT token. If it has a valid JWT Token then it sets the Authentication in the context, to specify that the current user is authenticated.

package com.javainuse.config;

import java.io.IOException;

import javax.servlet.FilterChain;

import javax.servlet.ServletException;

import javax.servlet.http.HttpServletRequest;

import javax.servlet.http.HttpServletResponse;

import org.springframework.beans.factory.annotation.Autowired;

import org.springframework.security.authentication.UsernamePasswordAuthenticationToken;

import org.springframework.security.core.context.SecurityContextHolder;

import org.springframework.security.core.userdetails.UserDetails;

import org.springframework.security.web.authentication.WebAuthenticationDetailsSource;

import org.springframework.stereotype.Component;

import org.springframework.web.filter.OncePerRequestFilter;

import com.javainuse.service.JwtUserDetailsService;

import io.jsonwebtoken.ExpiredJwtException;

@Component

public class JwtRequestFilter extends OncePerRequestFilter {

@Autowired

private JwtUserDetailsService jwtUserDetailsService;

@Autowired

private JwtTokenUtil jwtTokenUtil;

@Override

protected void doFilterInternal(HttpServletRequest request, HttpServletResponse response, FilterChain chain)

throws ServletException, IOException {

final String requestTokenHeader = request.getHeader("Authorization");

String username = null;

String jwtToken = null;

**// JWT Token is in the form "Bearer token". Remove Bearer word and get**

**// only the Token**

if (requestTokenHeader != null && requestTokenHeader.startsWith("Bearer ")) {

jwtToken = requestTokenHeader.substring(7);

try {

username = jwtTokenUtil.getUsernameFromToken(jwtToken);

} catch (IllegalArgumentException e) {

System.out.println("Unable to get JWT Token");

} catch (ExpiredJwtException e) {

System.out.println("JWT Token has expired");

}

} else {

logger.warn("JWT Token does not begin with Bearer String");

}

**// Once we get the token validate it.**

if (username != null && SecurityContextHolder.getContext().getAuthentication() == null) {

UserDetails userDetails = this.jwtUserDetailsService.loadUserByUsername(username);

**// if token is valid configure Spring Security to manually set**

**// authentication**

if (jwtTokenUtil.validateToken(jwtToken, userDetails)) {

UsernamePasswordAuthenticationToken usernamePasswordAuthenticationToken = new UsernamePasswordAuthenticationToken(

userDetails, null, userDetails.getAuthorities());

usernamePasswordAuthenticationToken

.setDetails(new WebAuthenticationDetailsSource().buildDetails(request));

**// After setting the Authentication in the context, we specify**

**// that the current user is authenticated. So it passes the**

**// Spring Security Configurations successfully.**

SecurityContextHolder.getContext().setAuthentication(usernamePasswordAuthenticationToken);

}

}

chain.doFilter(request, response);

}

}

* JwtAuthenticationEntryPoint

This class will extend Spring's AuthenticationEntryPoint class and override its method commence. It rejects every unauthenticated request and send error code 401

package com.javainuse.config;

import java.io.IOException;

import java.io.Serializable;

import javax.servlet.http.HttpServletRequest;

import javax.servlet.http.HttpServletResponse;

import org.springframework.security.core.AuthenticationException;

import org.springframework.security.web.AuthenticationEntryPoint;

import org.springframework.stereotype.Component;

@Component

public class JwtAuthenticationEntryPoint implements AuthenticationEntryPoint, Serializable {

private static final long serialVersionUID = -7858869558953243875L;

@Override

public void commence(HttpServletRequest request, HttpServletResponse response,

AuthenticationException authException) throws IOException {

response.sendError(HttpServletResponse.SC\_UNAUTHORIZED, "Unauthorized");

}

}

* WebSecurityConfig

This class extends the WebSecurityConfigurerAdapter is a convenience class that allows customization to both WebSecurity and HttpSecurity.

package com.javainuse.config;

import org.springframework.beans.factory.annotation.Autowired;

import org.springframework.context.annotation.Bean;

import org.springframework.context.annotation.Configuration;

import org.springframework.security.authentication.AuthenticationManager;

import org.springframework.security.config.annotation.authentication.builders.AuthenticationManagerBuilder;

import org.springframework.security.config.annotation.method.configuration.EnableGlobalMethodSecurity;

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.config.http.SessionCreationPolicy;

import org.springframework.security.core.userdetails.UserDetailsService;

import org.springframework.security.crypto.bcrypt.BCryptPasswordEncoder;

import org.springframework.security.crypto.password.PasswordEncoder;

import org.springframework.security.web.authentication.UsernamePasswordAuthenticationFilter;

@Configuration

@EnableWebSecurity

@EnableGlobalMethodSecurity(prePostEnabled = true)

public class WebSecurityConfig extends WebSecurityConfigurerAdapter {

@Autowired

private JwtAuthenticationEntryPoint jwtAuthenticationEntryPoint;

@Autowired

private UserDetailsService jwtUserDetailsService;

@Autowired

private JwtRequestFilter jwtRequestFilter;

@Autowired

public void configureGlobal(AuthenticationManagerBuilder auth) throws Exception {

**// configure AuthenticationManager so that it knows from where to load**

**// user for matching credentials**

**// Use BCryptPasswordEncoder**

auth.userDetailsService(jwtUserDetailsService).passwordEncoder(passwordEncoder());

}

@Bean

public PasswordEncoder passwordEncoder() {

return new BCryptPasswordEncoder();

}

@Bean

@Override

public AuthenticationManager authenticationManagerBean() throws Exception {

return super.authenticationManagerBean();

}

@Override

protected void configure(HttpSecurity httpSecurity) throws Exception {

**// We don't need CSRF for this example**

httpSecurity.csrf().disable()

**// dont authenticate this particular request**

.authorizeRequests().antMatchers**("/authenticate")**.permitAll().

**// all other requests need to be authenticated**

anyRequest().authenticated().and().

**// make sure we use stateless session; session won't be used to**

**// store user's state.**

exceptionHandling().authenticationEntryPoint(jwtAuthenticationEntryPoint).and().sessionManagement()

.sessionCreationPolicy(SessionCreationPolicy.STATELESS);

**// Add a filter to validate the tokens with every request**

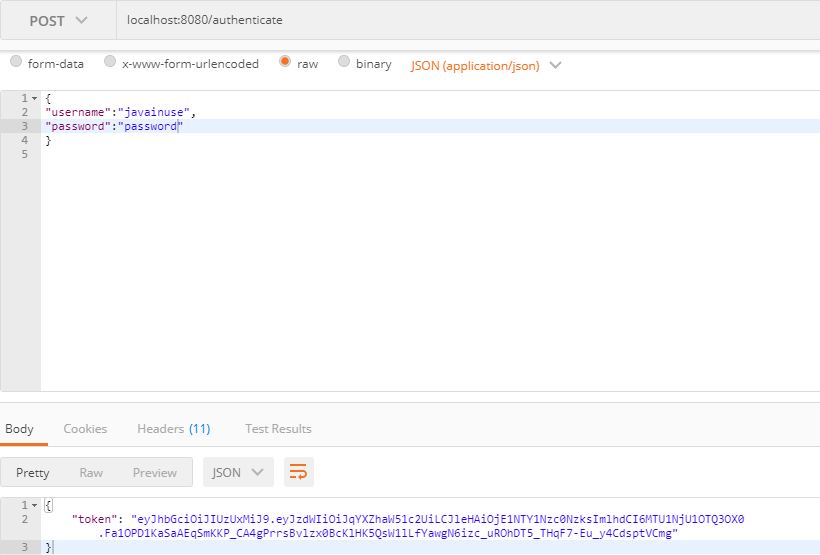
httpSecurity.addFilterBefore(jwtRequestFilter, UsernamePasswordAuthenticationFilter.class);

}

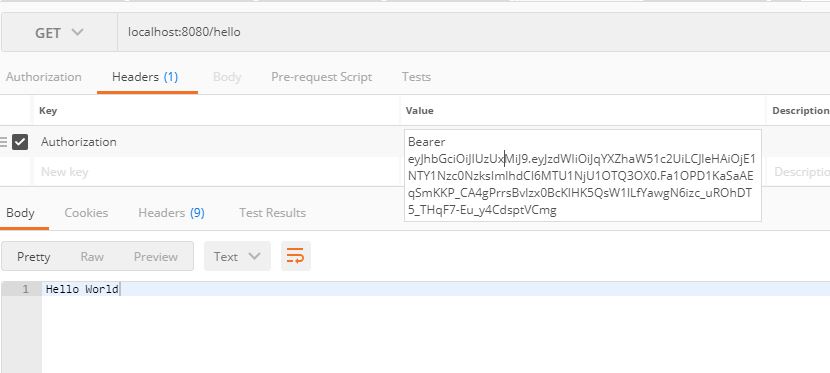
}

Start the Spring Boot Application

* Generate a JSON Web Token -

Create a POST request with url localhost:8080/authenticate. Body should have valid username and password. In our case username is javainuse and password is password.  


* Validate the JSON Web Token

- Try accessing the url localhost:8080/hello using the above generated token in the header as follows  


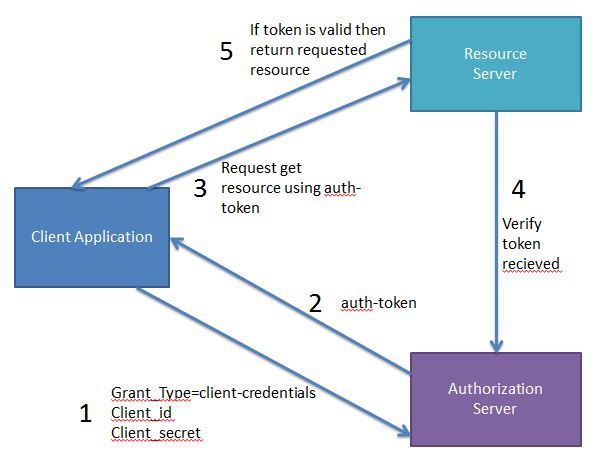
# **Spring Boot + OAuth 2 Client Credentials Grant - Hello World Example**

OAuth (Open Authorization) is a simple way to publish and interact with protected data.  
It is an open standard for token-based authentication and authorization on the Internet. It allows an end user's account information to be used by third-party services, such as Facebook, without exposing the user's password.  
When using OAuth2, grant type is the way an application gets the access token. Following are the grant types according to OAuth2 specification-

* Authorization code grant
* Implicit grant
* Resource owner credentials grant
* Client credentials grant
* Refresh token grant

In a previous series we had seen the [Authorization Code Grant](https://www.javainuse.com/spring/spring-boot-oauth-introduction) in detail.  
The Client Credentials Grant involves machine to machine authentication. Oauth usually consists of following actors -

* **Resource Owner(User) -** An entity capable of granting access to a protected resource. When the resource owner is a person, it is referred to as an end-user.
* **Client Application -** The machine that needs to be authenticated.
* **Authorization Server -** The server issuing access tokens to the client after successfully authenticating the resource owner and obtaining authorization
* **Resource Server -** The resource server is the OAuth 2.0 term for your API server. The resource server handles authenticated requests after the application has obtained an access token.

In case of Client credentials grant type the user has no role to play. As previously stated it is machine to machine communication. This is typically used by clients to access resources about themselves rather than to access a user's resources.  
  
This type of Authentication does not involve any end-user. Unlike Authorization Grant where the end user had to authenticate himself using Authorization Server like Gmail, here the machine it self authenticates itself to access a protected resource.

For example consider Trivago, a hotel aggregator portal which will be our client application.  
  
Trivago server will be accessing several third party APIs to show search results. Machine to machine authentication will be done by the Trivago server to access the third party API's to get the hotel data. Suppose it wants search data from makemytrip.com, so Trivago Server will authenticate itself by calling makemytrip's authorization server to get access token and then using this token access the makemytrip resource server to get the search result. So here-

* Client Application(Trivago Server) - Trivago Server which will need to get some reources from MakeMyTrip.com.
* Authorization Server(MakeMyTrip Authorization Server)- MakeMyTrip Authorization Server. Here Trivago should have already registered itself to the MakeMyTrip Authorization Server so that it can be authenticated and issued token.
* Resource Server(MakeMyTrip Resource Server) - MakeMyTrip application will then use the token it recieved from the Authorization Server to get resource from the MakeMyTrip Resource Server. MakeMyTrip ResourceServer will verify if the token recieved is valid by calling the Authorization server which issued it. If its valif it will return the requested resource

So 2 calls are required to be made by the client application to get the resource-

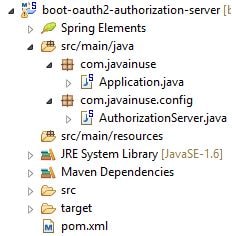
* Call to the Authorization Server to get the token.

|  |  |
| --- | --- |
| **Parameter** | **Value** |
| grant\_type (required) | client\_credentials |
| client\_id(required) | The client id |
| client\_secret(required) | The client secret key |

* After getting the token from the authorization server, the client application then needs to use this for getting resource from the resource server.

Lets Begin?

## Authorization Server

Lets first create the Authorization Server which will generate a token for client. Maven Project will be as follows-  
  


In the Maven we need the Spring oauth dependency. Maven will be as follows-

<?xml version="1.0" encoding="UTF-8"?>

<project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"

xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/maven-4.0.0.xsd">

<modelVersion>4.0.0</modelVersion>

<groupId>com.javainuse</groupId>

<artifactId>boot-oauth2-authorization-server</artifactId>

<version>0.0.1.SNAPSHOT</version>

<parent>

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

<artifactId>spring-boot-starter-parent</artifactId>

<version>1.3.0.RELEASE</version>

</parent>

<dependencies>

<dependency>

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

<artifactId>spring-boot-starter-web</artifactId>

</dependency>

<dependency>

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

<artifactId>spring-boot-starter-actuator</artifactId>

</dependency>

<dependency>

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

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

</dependency>

**<dependency>**

**<groupId>org.springframework.security.oauth</groupId>**

**<artifactId>spring-security-oauth2</artifactId>**

**</dependency>**

</dependencies>

<build>

<plugins>

<plugin>

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

<artifactId>spring-boot-maven-plugin</artifactId>

</plugin>

</plugins>

</build>

</project>

Define the Spring Boot Main Application.

package com.javainuse;

import java.security.Principal;

import org.springframework.boot.SpringApplication;

import org.springframework.boot.autoconfigure.SpringBootApplication;

import org.springframework.security.oauth2.config.annotation.web.configuration.EnableResourceServer;

import org.springframework.web.bind.annotation.RequestMapping;

import org.springframework.web.bind.annotation.RestController;

@RestController

@EnableResourceServer

@SpringBootApplication

public class Application {

public static void main(String[] args) {

SpringApplication.run(Application.class, args);

}

//This method will be used to check if the user has a valid token to access the resource

@RequestMapping("/validateUser")

public Principal user(Principal user) {

return user;

}

}

Configure the Authorization Server. The @EnableAuthorizationServer annotation is used to configure the OAuth 2.0 Authorization Server mechanism and defines the behaviour of various endpoints when interacting with the authorization server.

package com.javainuse.config;

import org.springframework.beans.factory.annotation.Autowired;

import org.springframework.context.annotation.Configuration;

import org.springframework.security.authentication.AuthenticationManager;

import org.springframework.security.oauth2.config.annotation.configurers.ClientDetailsServiceConfigurer;

import org.springframework.security.oauth2.config.annotation.web.configuration.AuthorizationServerConfigurerAdapter;

import org.springframework.security.oauth2.config.annotation.web.configuration.EnableAuthorizationServer;

import org.springframework.security.oauth2.config.annotation.web.configurers.AuthorizationServerEndpointsConfigurer;

@Configuration

@EnableAuthorizationServer

public class AuthorizationServer extends AuthorizationServerConfigurerAdapter {

@Autowired

private AuthenticationManager authenticationManager;

@Override

public void configure(AuthorizationServerEndpointsConfigurer endpoints) throws Exception {

endpoints.authenticationManager(authenticationManager);

}

@Override

public void configure(ClientDetailsServiceConfigurer clients) throws Exception {

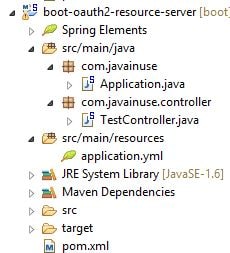
clients.inMemory().withClient("javainuse-client").secret("javainuse-secret")

**.authorizedGrantTypes("client\_credentials")**.scopes("resource-server-read", "resource-server-write");

}

}

## Resource Server

Next create the Resource Server which has the reource to be accessed by the client. Maven Project will be as follows-  
  


In the Maven we need the Spring oauth dependency. Maven will be as follows-

<?xml version="1.0" encoding="UTF-8"?>

<project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"

xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/maven-4.0.0.xsd">

<modelVersion>4.0.0</modelVersion>

<groupId>com.javainuse</groupId>

<artifactId>boot-oauth2-resource-server</artifactId>

<version>0.0.1.SNAPSHOT</version>

<parent>

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

<artifactId>spring-boot-starter-parent</artifactId>

<version>1.3.0.RELEASE</version>

</parent>

<dependencies>

<dependency>

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

<artifactId>spring-boot-starter-web</artifactId>

</dependency>

<dependency>

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

<artifactId>spring-boot-starter-actuator</artifactId>

</dependency>

<dependency>

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

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

</dependency>

**<dependency>**

**<groupId>org.springframework.security.oauth</groupId>**

**<artifactId>spring-security-oauth2</artifactId>**

**</dependency>**

</dependencies>

<build>

<plugins>

<plugin>

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

<artifactId>spring-boot-maven-plugin</artifactId>

</plugin>

</plugins>

</build>

</project>

Define the Spring Boot Main Application. Configure the Resource Serverusing @EnableResourceServer annotation. It means the service expects an access token in order to process the request. Access token should be obtained from Authorization Server by OAuth 2.0 Client before calling the Resource Server.

package com.javainuse;

import org.springframework.boot.SpringApplication;

import org.springframework.boot.autoconfigure.SpringBootApplication;

import org.springframework.security.oauth2.config.annotation.web.configuration.EnableResourceServer;

import org.springframework.web.bind.annotation.RestController;

@EnableResourceServer

@SpringBootApplication

public class Application {

public static void main(String[] args) {

SpringApplication.run(Application.class, args);

}

}

Define the Controller, to expose API which can be accessed only using valid token.

package com.javainuse.controller;

import org.springframework.web.bind.annotation.RequestMapping;

import org.springframework.web.bind.annotation.RestController;

@RestController

public class TestController {

@RequestMapping("/test")

public String test() {

return "Hello World";

}

}

Next define the url of the authorization server to be called by the resource server for verifying the token in the application.yml as follows.

security:

oauth2:

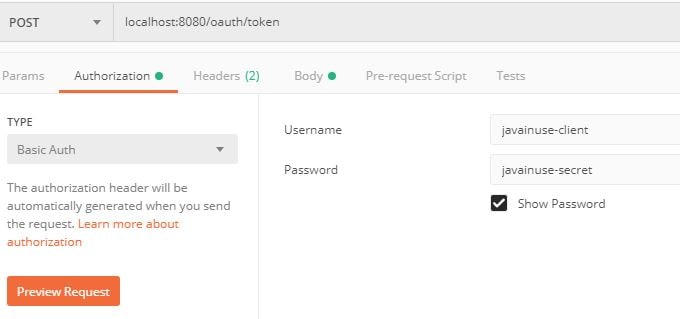
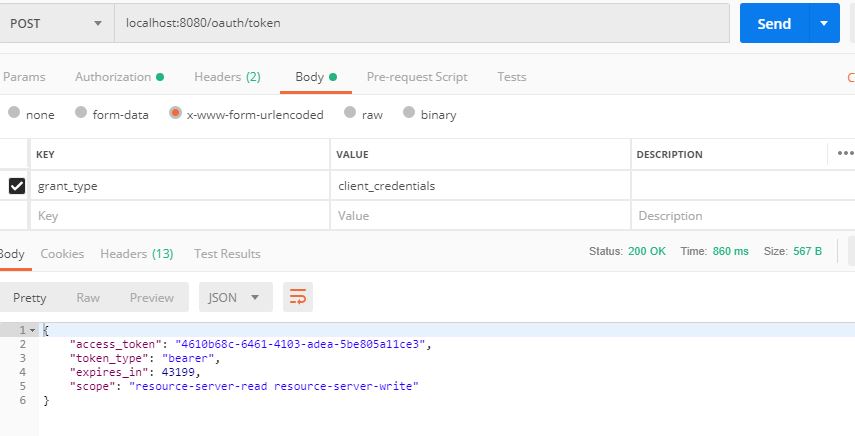
resource:

userInfoUri: http://localhost:8080/validateUser

server:

port: 9090

Finally start the Authorization Server and the Resource Server.

* First get the Access Token by making a POST request to localhost:8080/oauth/token
  + Specify the client\_id and client\_secret in the header using base64 encoding.  
    
  + Next specify the grant type as Client Credentials in body and send the request.  
      
    We get the token as response
* Get the Resource using the access token received above and making a GET call to localhost:9090/test.  
  The token is specified as Authorization Bearer.  
  