Microservices with Spring Boot — Authentication with JWT (Part 3)

Filter, Validate, and Generate Tokens

OMAR ELGABRY Follow
Jun 11, 2018 · 5 min read

Authentication with JWT

> *The Github repository for the application:*
> *https://github.com/OmarElGabry/microservices-spring-boot*

. . .

Authentication Workflow

The authentication flow is simple as:

1. The user sends a request to get a token passing his credentials.
2. The server validates the credentials and sends back a token.

3. With every request, the user has to provide the token, and server will validate that token.

We'll introduce another service called 'auth service' for validating user credentials, and issuing tokens.

What about validating the token? Well, it can be implemented in the auth service itself, and the gateway has to call the auth service to validate the token before allowing the requests to go to any service.

Instead, we can validate the tokens at the gateway level, and let the auth service validate user credentials, and issue tokens. And that's what we're going to do here.

In both ways, we are blocking the requests unless it's authenticated (except the requests for generating tokens).

JSON Based Token (JWT)

A token is an encoded string, generated by our application (after being authenticated) and sent by the user along each request to allow access to the resources exposed by our application.

JSON Based Token (JWT) is a JSON-based open standard for creating access tokens. It consists of three parts; header, payload, and signature.

The header contains the hashing algorithm

```
{type: "JWT", hash: "HS256"}
```

The payload contains attributes (username, email, etc) and their values.

```
{username: "Omar", email: "omar@example.com", admin: true }
```

The signature is hashing of: `Header + "." + Payload + Secret key`

Gateway

In the gateway, we need to do two things: (1) validate tokens with every request, and (2) prevent all unauthenticated requests to our services. Fair enough?

In the `pom.xml` add spring security and JWT dependencies.

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

<dependency>
  <groupId>io.jsonwebtoken</groupId>
  <artifactId>jjwt</artifactId>
  <version>0.9.0</version>
</dependency>
```

In `application.properties` add paths to auth service (we'll create it later).

```
# Map path to auth service
zuul.routes.auth-service.path=/auth/**
zuul.routes.auth-service.service-id=AUTH-SERVICE

# By default, all requests to gallery service for example will start with: "/gallery/"
# What will be sent to the gallery service is what comes after the path defined,
# So, if request is "/gallery/view/1", gallery service will get "/view/1".
# In case of auth, we need to pass the "/auth/" in the path to auth service. So, set strip-prefi
zuul.routes.auth-service.strip-prefix=false

# Exclude authorization from sensitive headers
zuul.routes.auth-service.sensitive-headers=Cookie,Set-Cookie
```

application.properties hosted with ♥ by GitHub view raw

To define our security configurations, create a class, and annotated with `@EnableWebSecurity`, and extends `WebSecurityConfigurerAdapter` class to override and provide our own custom security configurations.

```
package com.eureka.zuul.security;

import javax.servlet.http.HttpServletResponse;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.context.annotation.Bean;
import org.springframework.http.HttpMethod;
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.WebSecurityConfigurerAda
import org.springframework.security.config.http.SessionCreationPolicy;
import org.springframework.security.web.authentication.UsernamePasswordAuthenticationFilter;

import com.eureka.zuul.security.JwtConfig;

@EnableWebSecurity      // Enable security config. This annotation denotes config for spring sec
```

```
public class SecurityTokenConfig extends WebSecurityConfigurerAdapter {
        @Autowired
        private JwtConfig jwtConfig;

        @Override
        protected void configure(HttpSecurity http) throws Exception {
           http
                .csrf().disable()
                    // make sure we use stateless session; session won't be used to store user's
                    .sessionManagement().sessionCreationPolicy(SessionCreationPolicy.STATELESS)
                .and()
                    // handle an authorized attempts
                    .exceptionHandling().authenticationEntryPoint((req, rsp, e) -> rsp.sendError
                .and()
                   // Add a filter to validate the tokens with every request
                   .addFilterAfter(new JwtTokenAuthenticationFilter(jwtConfig), UsernamePassword
                // authorization requests config
                .authorizeRequests()
                   // allow all who are accessing "auth" service
                   .antMatchers(HttpMethod.POST, jwtConfig.getUri()).permitAll()
                   // must be an admin if trying to access admin area (authentication is also re
                   .antMatchers("/gallery" + "/admin/**").hasRole("ADMIN")
                   // Any other request must be authenticated
                   .anyRequest().authenticated();
        }

        @Bean
        public JwtConfig jwtConfig() {
           return new JwtConfig();
        }
}
```

SecurityTokenConfig.java hosted with ♥ by GitHub view raw

Spring has filters that will get executed within the life-cycle of the request (filter chain). To enable and use these filters, we need to extend the class of any of these filters.

By default spring will try to figure out when the filter should be executed. Otherwise, we can also define when should be executed (after or before another filter).

The `JwtConfig` is just a class contains configuration variables.

```
public class JwtConfig {
    @Value("${security.jwt.uri:/auth/**}")
    private String Uri;

    @Value("${security.jwt.header:Authorization}")
    private String header;
```

```
    @Value("${security.jwt.prefix:Bearer }")
    private String prefix;

    @Value("${security.jwt.expiration:#{24*60*60}}")
    private int expiration;

    @Value("${security.jwt.secret:JwtSecretKey}")
    private String secret;

    // getters ...
}
```

JwtConfig.java hosted with ♥ by GitHub view raw

The last step is to implement our filter that validates the tokens. We're using `OncePerRequestFilter`. It guarantee a single execution per request (since you can have a filter on the filter chain more than once).

```
package com.eureka.zuul.security;

import java.io.IOException;
import java.util.List;
import java.util.stream.Collectors;

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

import org.springframework.security.authentication.UsernamePasswordAuthenticationToken;
import org.springframework.security.core.authority.SimpleGrantedAuthority;
import org.springframework.security.core.context.SecurityContextHolder;
import org.springframework.web.filter.OncePerRequestFilter;

import com.eureka.zuul.security.JwtConfig;

import io.jsonwebtoken.Claims;
import io.jsonwebtoken.Jwts;

public class JwtTokenAuthenticationFilter extends  OncePerRequestFilter {

        private final JwtConfig jwtConfig;

        public JwtTokenAuthenticationFilter(JwtConfig jwtConfig) {
                this.jwtConfig = jwtConfig;
        }

        @Override
        protected void doFilterInternal(HttpServletRequest request, HttpServletResponse response
```

```
32                  throws ServletException, IOException {
33
34          // 1. get the authentication header. Tokens are supposed to be passed in the aut
35          String header = request.getHeader(jwtConfig.getHeader());
36
37          // 2. validate the header and check the prefix
38          if(header == null || !header.startsWith(jwtConfig.getPrefix())) {
39                  chain.doFilter(request, response);              // If not valid, go to t
40                  return;
41          }
42
43          // If there is no token provided and hence the user won't be authenticated.
44          // It's Ok. Maybe the user accessing a public path or asking for a token.
45
46          // All secured paths that needs a token are already defined and secured in confi
47          // And If user tried to access without access token, then he won't be authentica
48
49          // 3. Get the token
50          String token = header.replace(jwtConfig.getPrefix(), "");
51
52          try {   // exceptions might be thrown in creating the claims if for example the
53
54                  // 4. Validate the token
55                  Claims claims = Jwts.parser()
56                                  .setSigningKey(jwtConfig.getSecret().getBytes())
57                                  .parseClaimsJws(token)
58                                  .getBody();
59
60                  String username = claims.getSubject();
61                  if(username != null) {
62                          @SuppressWarnings("unchecked")
63                          List<String> authorities = (List<String>) claims.get("authoritie
64
65                          // 5. Create auth object
66                          // UsernamePasswordAuthenticationToken: A built-in object, used
67                          // It needs a list of authorities, which has type of GrantedAuth
68                           UsernamePasswordAuthenticationToken auth = new UsernamePassword
69                                                          username, null, authorities.str
70
71                           // 6. Authenticate the user
72                           // Now, user is authenticated
73                           SecurityContextHolder.getContext().setAuthentication(auth);
74                  }
75
76          } catch (Exception e) {
77                  // In case of failure. Make sure it's clear; so guarantee user won't be
78                  SecurityContextHolder.clearContext();
79          }
80
81          // go to the next filter in the filter chain
82          chain.doFilter(request, response);
```

```
        }

}
```

JwtTokenAuthenticationFilter.java hosted with ♥ by GitHub view raw

Auth Service

In the auth service, we need to (1) validate the user credentials, and if valid, (2) generate a token, otherwise, throw an exception.

In the `pom.xml` add the following dependencies: Web, Eureka Client, Spring Security and JWT.

```
....
 <dependencies>
         <dependency>
                <groupId>org.springframework.boot</groupId>
                <artifactId>spring-boot-starter-web</artifactId>
        </dependency>
         <dependency>
                 <groupId>org.springframework.cloud</groupId>
                 <artifactId>spring-cloud-starter-netflix-eureka-client</artifactId>
         </dependency>
         <dependency>
                <groupId>org.springframework.boot</groupId>
                <artifactId>spring-boot-starter-security</artifactId>
        </dependency>
        <dependency>
                <groupId>io.jsonwebtoken</groupId>
                <artifactId>jjwt</artifactId>
                <version>0.9.0</version>
         </dependency>
         <dependency>
                <groupId>org.springframework.boot</groupId>
                <artifactId>spring-boot-devtools</artifactId>
                <optional>true</optional>
         </dependency>
</dependencies>
....
```

pom.xml hosted with ♥ by GitHub view raw

In the `application.properties`

```
spring.application.name=auth-service
server.port=9100
eureka.client.service-url.default-zone=http://localhost:8761/eureka
```

application.properties hosted with ♥ by GitHub view raw

As we did in the Gateway for security configurations, create a class, annotated with `@EnableWebSecurity`, and extends `WebSecurityConfigurerAdapter`

```
1  package com.eureka.auth.security;
2
3  import javax.servlet.http.HttpServletResponse;
4
5  import org.springframework.beans.factory.annotation.Autowired;
6  import org.springframework.context.annotation.Bean;
7  import org.springframework.http.HttpMethod;
8  import org.springframework.security.config.annotation.authentication.builders.AuthenticationMana
9  import org.springframework.security.config.annotation.web.builders.HttpSecurity;
10 import org.springframework.security.config.annotation.web.configuration.EnableWebSecurity;
11 import org.springframework.security.config.annotation.web.configuration.WebSecurityConfigurerAda
12 import org.springframework.security.config.http.SessionCreationPolicy;
13 import org.springframework.security.core.userdetails.UserDetailsService;
14 import org.springframework.security.crypto.bcrypt.BCryptPasswordEncoder;
15
16 import com.eureka.auth.security.JwtConfig;
17
18 @EnableWebSecurity      // Enable security config. This annotation denotes config for spring sec
19 public class SecurityCredentialsConfig extends WebSecurityConfigurerAdapter {
20
21         @Autowired
22         private UserDetailsService userDetailsService;
23
24         @Autowired
25         private JwtConfig jwtConfig;
26
27         @Override
28         protected void configure(HttpSecurity http) throws Exception {
29                 http
30                     .csrf().disable()
31                      // make sure we use stateless session; session won't be used to store user'
32                     .sessionManagement().sessionCreationPolicy(SessionCreationPolicy.STATELESS)
33                 .and()
34                     // handle an authorized attempts
35                     .exceptionHandling().authenticationEntryPoint((req, rsp, e) -> rsp.sendError
36                 .and()
37                     // Add a filter to validate user credentials and add token in the response h
38
39                     // What's the authenticationManager()?
40                     // An object provided by WebSecurityConfigurerAdapter, used to authenticate
41                     // The filter needs this auth manager to authenticate the user.
42                     .addFilter(new JwtUsernameAndPasswordAuthenticationFilter(authenticationMana
43                 .authorizeRequests()
44                     // allow all POST requests
```

```
                    .antMatchers(HttpMethod.POST, jwtConfig.getUri()).permitAll()
                    // any other requests must be authenticated
                    .anyRequest().authenticated();
        }

        // Spring has UserDetailsService interface, which can be overriden to provide our implem
        // The UserDetailsService object is used by the auth manager to load the user from datab
        // In addition, we need to define the password encoder also. So, auth manager can compar
        @Override
        protected void configure(AuthenticationManagerBuilder auth) throws Exception {
                auth.userDetailsService(userDetailsService).passwordEncoder(passwordEncoder());
        }

        @Bean
        public JwtConfig jwtConfig() {
                return new JwtConfig();
        }

        @Bean
        public BCryptPasswordEncoder passwordEncoder() {
            return new BCryptPasswordEncoder();
        }
}
```

SecurityCredentialsConfig.java hosted with ♥ by GitHub view raw

As you can see in the code above, we need to implement `UserDetailsService` interface.

This class acts like a provider for the user; meaning it loads the user from the database (or any data source). It doesn't do authentication. It just loads the user given his username.

```
package com.eureka.auth.security;

import java.util.Arrays;
import java.util.List;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.security.core.GrantedAuthority;
import org.springframework.security.core.authority.AuthorityUtils;
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.security.crypto.bcrypt.BCryptPasswordEncoder;
import org.springframework.stereotype.Service;

@Service   // It has to be annotated with @Service.
public class UserDetailsServiceImpl implements UserDetailsService  {
```

```java

	@Autowired
	private BCryptPasswordEncoder encoder;

	@Override
	public UserDetails loadUserByUsername(String username) throws UsernameNotFoundException

		// hard coding the users. All passwords must be encoded.
		final List<AppUser> users = Arrays.asList(
			new AppUser(1, "omar", encoder.encode("12345"), "USER"),
			new AppUser(2, "admin", encoder.encode("12345"), "ADMIN")
		);


		for(AppUser appUser: users) {
			if(appUser.getUsername().equals(username)) {

				// Remember that Spring needs roles to be in this format: "ROLE_
				// So, we need to set it to that format, so we can verify and co
				List<GrantedAuthority> grantedAuthorities = AuthorityUtils
					.commaSeparatedStringToAuthorityList("ROLE_" + appUser.g

				// The "User" class is provided by Spring and represents a model
				// And used by auth manager to verify and check user authenticat
				return new User(appUser.getUsername(), appUser.getPassword(), gr
			}
		}

		// If user not found. Throw this exception.
		throw new UsernameNotFoundException("Username: " + username + " not found");
	}

	// A (temporary) class represent the user saved in the database.
	private static class AppUser {
		private Integer id;
		private String username, password;
		private String role;

		public AppUser(Integer id, String username, String password, String role) {
			this.id = id;
			this.username = username;
			this.password = password;
			this.role = role;
		}

		// getters and setters ....
	}
}
```

And here comes the last step; the filter.

We're using `JwtUsernameAndPasswordAuthenticationFilter`. It's used to validate user credentials, and generate tokens. The username and password must be sent in a POST request.

```
package com.eureka.auth.security;

import java.io.IOException;
import java.sql.Date;
import java.util.Collections;
import java.util.stream.Collectors;

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.GrantedAuthority;
import org.springframework.security.web.authentication.UsernamePasswordAuthenticationFilter;
import org.springframework.security.web.util.matcher.AntPathRequestMatcher;

import com.eureka.auth.security.JwtConfig;
import com.fasterxml.jackson.databind.ObjectMapper;

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

public class JwtUsernameAndPasswordAuthenticationFilter extends UsernamePasswordAuthenticationFi

        // We use auth manager to validate the user credentials
        private AuthenticationManager authManager;

        private final JwtConfig jwtConfig;

        public JwtUsernameAndPasswordAuthenticationFilter(AuthenticationManager authManager, Jwt
                this.authManager = authManager;
                this.jwtConfig = jwtConfig;

                // By default, UsernamePasswordAuthenticationFilter listens to "/login" path.
                // In our case, we use "/auth". So, we need to override the defaults.
                this.setRequiresAuthenticationRequestMatcher(new AntPathRequestMatcher(jwtConfig
        }

        @Override
        public Authentication attemptAuthentication(HttpServletRequest request, HttpServletResp
```

```
	public Authentication attemptAuthentication(HttpServletRequest request, HttpServletRespo
			throws AuthenticationException {

		try {

			// 1. Get credentials from request
			UserCredentials creds = new ObjectMapper().readValue(request.getInputStr

			// 2. Create auth object (contains credentials) which will be used by au
			UsernamePasswordAuthenticationToken authToken = new UsernamePasswordAuth
					creds.getUsername(), creds.getPassword(), Collections.em

			// 3. Authentication manager authenticate the user, and use UserDetialsS
			return authManager.authenticate(authToken);

		} catch (IOException e) {
			throw new RuntimeException(e);
		}
	}

	// Upon successful authentication, generate a token.
	// The 'auth' passed to successfulAuthentication() is the current authenticated user.
	@Override
	protected void successfulAuthentication(HttpServletRequest request, HttpServletResponse
			Authentication auth) throws IOException, ServletException {

		Long now = System.currentTimeMillis();
		String token = Jwts.builder()
			.setSubject(auth.getName())
			// Convert to list of strings.
			// This is important because it affects the way we get them back in the
			.claim("authorities", auth.getAuthorities().stream()
				.map(GrantedAuthority::getAuthority).collect(Collectors.toList()
			.setIssuedAt(new Date(now))
			.setExpiration(new Date(now + jwtConfig.getExpiration() * 1000))  // in
			.signWith(SignatureAlgorithm.HS512, jwtConfig.getSecret().getBytes())
			.compact();

		// Add token to header
		response.addHeader(jwtConfig.getHeader(), jwtConfig.getPrefix() + token);
	}

	// A (temporary) class just to represent the user credentials
	private static class UserCredentials {
	    private String username, password;
	    // getters and setters ...
	}
}

```

Common Service

When you have common configuration variables, enum classes, or logic, used by multiple services, like the one we had `JwtConfig`. Instead of duplicating the code, we put it in a separate service that can be included and used as a dependency in other services.

To do so, just create a new project (service), call it 'common', and follow the same steps as we did with the image service. So, In `pom.xml` file

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

<dependency>
  <groupId>org.springframework.cloud</groupId>
  <artifactId>spring-cloud-starter-netflix-eureka-client</artifactId>
</dependency>
```

In the `application.properties`

```
spring.application.name=common-service
server.port=9200
eureka.client.service-url.default-zone=http://localhost:8761/eureka
```

In the spring boot main application class

```
package com.eureka.common;

import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;
import org.springframework.cloud.netflix.eureka.EnableEurekaClient;

@SpringBootApplication
@EnableEurekaClient
public class SpringEurekaCommonApp {

public static void main(String[] args) {
  SpringApplication.run(SpringEurekaCommonApp.class, args);
 }
}
```

Then, copy `JwtConfig` class we created earlier in Gateway in common service.

```
package com.eureka.common.security;
```

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

public class JwtConfig {
   // ...
}
```

Now, to be able to call `JwtConfig` class from other services, like auth and gateway, we just need to add the common service in `pom.xml` as dependency.

```
<dependency>
  <groupId>com.eureka.common</groupId>
  <artifactId>spring-eureka-common</artifactId>
  <version>0.0.1-SNAPSHOT</version>
</dependency>
```

And In our auth and gateway service …

```
// change these lines of code
import com.eureka.zuul.security.JwtConfig;
import com.eureka.auth.security.JwtConfig;

// to reference the class in common service instead
import com.eureka.common.security.JwtConfig;
```

Testing our Microservices

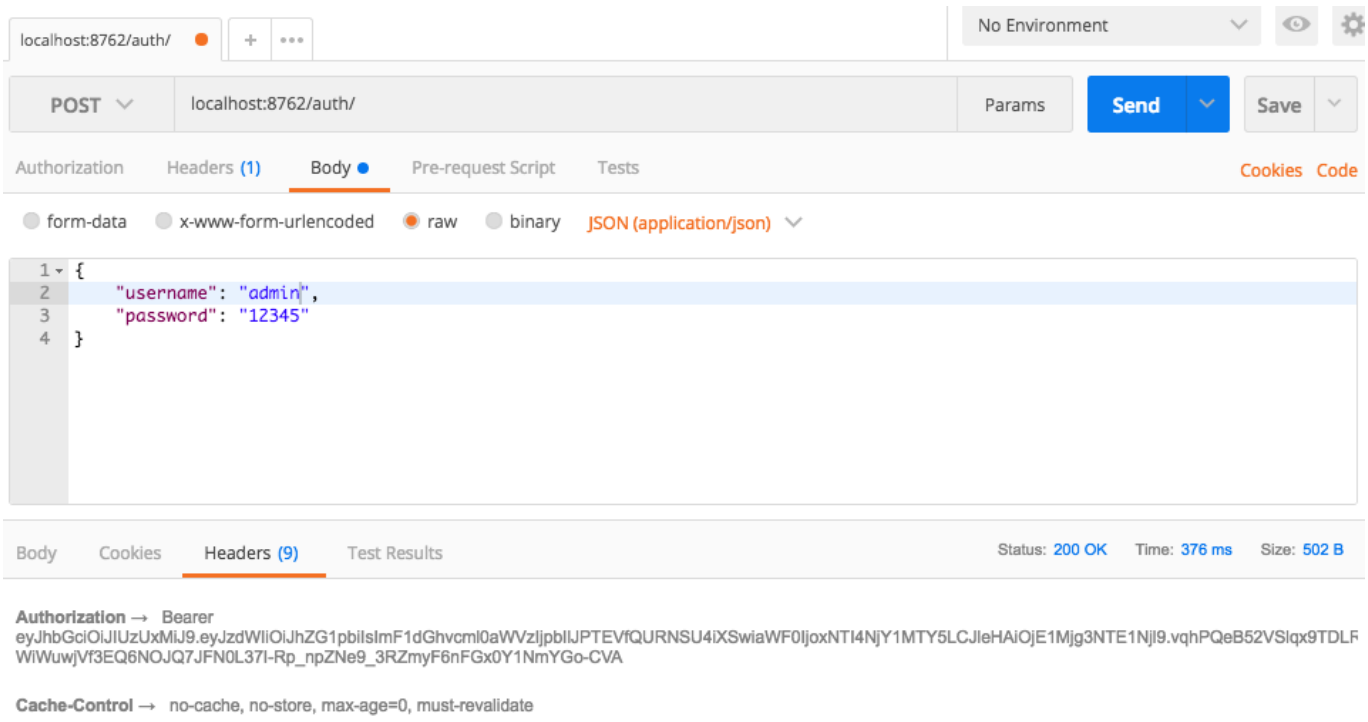
Now we plugged in the authentication logic, we can validate credentials, issue tokens, and authenticate our users seamlessly.

So, run our Eureka Server. Then, run other services: image, gallery, common, auth, and finally, the gateway.

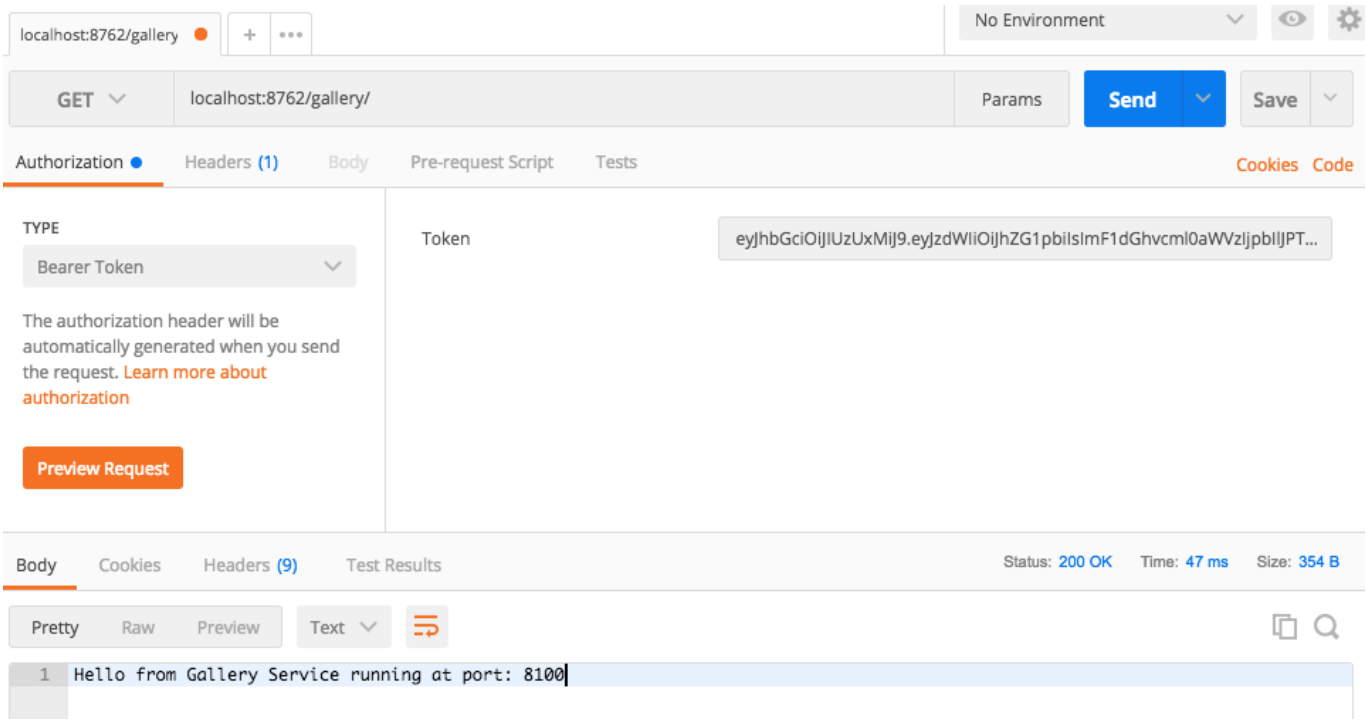
First, let's try to access gallery service `localhost:8762/gallery` without a token. You should get *Unauthorized* error.

```
{
    "timestamp": "...",
    "status": 401,
    "error": "Unauthorized",
    "message": "No message available",
    "path": "/gallery/"
}
```

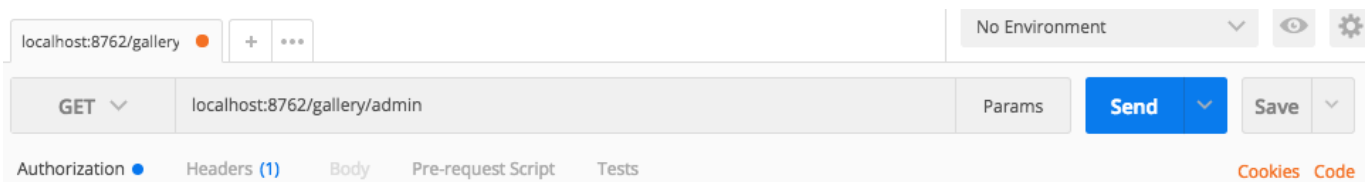
To get a token, send user credentials to `localhost:8762/auth` (we hardcoded two users in `UserDetailsServiceImpl` class above), and make sure the `Content-Type` in the headers is assigned to `application/json`

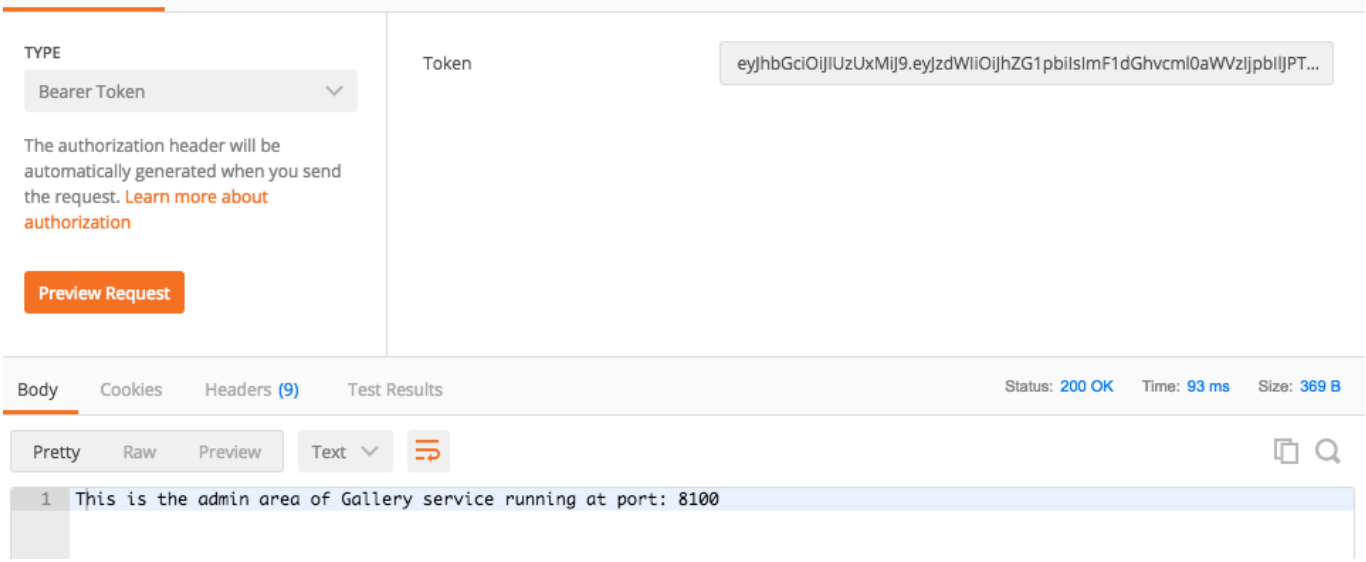


Now, we can make a request to gallery service passing the token in the header.



If token was created for the admin user, then you should be able to access admin area of gallery service.





Again, if you are running multiple instances of gallery service, each running at a different port, then requests will be distributed equally across them.

. . .

Thank you for reading! If you enjoyed it, please clap for it.