Spring Cloud Security with Netflix Zuul

Bharat Raj Meriyala
Dec 2, 2018 · 5 min read

Microservices and cloud micro-functions are the 2 best things to happen in computer programming history since C programming. Both of these are changing the way we design and architect our systems for high availability, fault tolerance, smaller DevOps cycles and of course for following the basic principles of following KISS(Keep it simple stupid) in computer programming.

Here I will touch upon, managing the security of a microservice based application using Netflix zuul. The entire code for this article can be found at this Github link:

BharatRajMeriyala/SpringCloudSecurity

Contribute to BharatRajMeriyala/SpringCloudSecurity...

github.com

Any microservices design these days requires us to have multiple individual services, that are designed to a specific task. Among these following are very important:

1. A load balancer: This piece of software is generally responsible for distributing the load among multiple endpoints, without we hardcoding the endpoints. For example if we have 2 endpoints as http://localhost:8080 and http://localhost:8081, without a load balancer, we would have to hardcode this list somewhere. With a load balancer in place we wouldn't have to do that.Ribbon, is one of the famous Spring security solution for load balancing.

2. A network discoverer: For a load balancer to automatically route all its requests to endpoints, we would want a network discovery software. The network discoverer would usually register any new instance coming up. The load balancer can then contact the network discoverer to find out the available endpoints and route its

requests based on a round robin mechanism. Eureka is the most preferred network discoverer and Ribbon is designed to work with it seamlessly, when used with Feign.

3. API Gateway: API Gateway is a critical piece of software required to route requests inside our microservices network. An API Gateway like zuul, sits on top of Eureka, where it registers itself and is available for all the requests.

A simple illustration of zuul Gateway is as follows:

Assuming, zuul is running at http://localhost:8765, a typical request to zuul gateway url scheme looks like this:

<Gateway-URL>/<Service-name>/handler-path

The above scheme can be understood with following example:

Say we have hosted a microservice(my-microsvc) that works for a following URL:

http://localhost:8100/basepath

Then the same can be called with zuul gateway like this:

http://localhost:8765/my-microsvc/basepath

So we are appending the microservice name in the URL.

This is amazing because this way we can solve all our inter microservice communication through Gateway, without knowing a lot of details. An API Gateway can also be used for advanced features like Rate-limiting, shared sessions or security as well. In this article we see more of this security that can be possible.

A beautiful Udemy course is available for those who want more details and see things in action:

https://www.udemy.com/microservices-with-spring-boot-and-spring-cloud/learn/v4/overview

Spring Security with Zuul

A suggested before, I have already placed the code in github repo. I would suggest to clone the repo and run the services one by one before going ahead.

api-gateway service

gateway.Application → We are enabling the zuul proxy configuration here

```
@SpringBootApplication
@EnableZuulProxy
public class Application {

    public static void main(String[] args) {

        SpringApplication.run(Application.class, args);
    }
}
```

gateway.SecurityConfig → Following the general principles of spring security, lets have a Spring Security configuration by extending the `WebSecurityConfigurerAdapter` base class and overriding individual methods. In this case we will override configure mehod.

```
@EnableWebSecurity
public class SecurityConfig extends WebSecurityConfigurerAdapter {

    @Autowired
    private JwtAuthenticationConfig config;

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

    @Override
    protected void configure(HttpSecurity httpSecurity) throws Exception {
        httpSecurity
                .csrf().disable()
                .logout().disable()
                .formLogin().disable()
                .sessionManagement().sessionCreationPolicy(SessionCreationPolicy.STATELESS)
                .and()
                    .anonymous()
                .and()
                    .exceptionHandling().authenticationEntryPoint(
                            (req, rsp, e) -> rsp.sendError(HttpServletResponse.SC_UNAUTHORIZED))
                .and()
                    .addFilterAfter(new JwtTokenAuthenticationFilter(config),
                            UsernamePasswordAuthenticationFilter.class)
                .authorizeRequests()
                    .antMatchers(config.getUrl()).permitAll()
                    .antMatchers("/backend/admin").hasRole("ADMIN")
                    .antMatchers("/backend/user").hasRole("USER")
                    .antMatchers("/backend/guest").permitAll();
    }
}
```

All we are doing here is basically, allowing a passthrough for /login URL and forcing the authentication of other URL. The important thing to note here is presence of addFilterAfter.

The addFilterAfter will basically, ensure JwtTokenAuthenticationFilter is executed before performing any validation, including the /login we are going to talk about.

```
public class JwtTokenAuthenticationFilter extends OncePerRequestFilter {
```

```
    private final JwtAuthenticationConfig config;

    public JwtTokenAuthenticationFilter(JwtAuthenticationConfig config) {
        this.config = config;
    }

    @Override
    protected void doFilterInternal(HttpServletRequest req, HttpServletResponse rsp, FilterChain filterChain)
            throws ServletException, IOException {
        String token = req.getHeader(config.getHeader());
        if (token != null && token.startsWith(config.getPrefix() + " ")) {
            token = token.replace(config.getPrefix() + " ", "");
            try {
                Claims claims = Jwts.parser()
                        .setSigningKey(config.getSecret().getBytes())
                        .parseClaimsJws(token)
                        .getBody();
                String username = claims.getSubject();
                @SuppressWarnings("unchecked")
                List<String> authorities = claims.get("authorities", List.class);
                if (username != null) {
                    UsernamePasswordAuthenticationToken auth = new UsernamePasswordAuthenticationToken(username, null,
                            authorities.stream().map(SimpleGrantedAuthority::new).collect(Collectors.toList()));
                    SecurityContextHolder.getContext().setAuthentication(auth);
                }
            } catch (Exception ignore) {
                SecurityContextHolder.clearContext();
            }
        }
        filterChain.doFilter(req, rsp);
    }
}
```

The above filter basically does 3 things:

1. Parse the JWT present inside the request header. I have used jjwt from Maven Central for JWT tokens.
2. Fetch the claims present inside the parsed JWT token.
3. Finally set the security context so that it is available for the requests downstream.

The most important thing to note in the above, configure method is that we are allowing /login to pass through without any authentication. This is logical because we do not want the token authentication to pitch in at this point of time. And performing null checks at the filter level to ensure we do not get exception for case of /login.

for /login to reach the other service auth-center, we will define the following in application.yml file:

```
zuul:
  routes:
    auth-center:
      path: /login/**
      url: http://127.0.0.1:8081/
      sensitiveHeaders: Cookie,Set-Cookie
      stripPrefix: false
    backend-service:
      path: /backend/**
      url: http://127.0.0.1:8082/

raj.security.jwt.secret: asupersecretpassword
```

auth-center

This service is designed basically to generate the auth tokens after validating the username and password passed.

As usual, we will define a security configurer that looks like below:

```
@Autowired
public void configureGlobal(AuthenticationManagerBuilder auth) throws Exception {
    auth.inMemoryAuthentication()
            .withUser("admin").password("admin").roles("ADMIN", "USER").and()
            .withUser("raj").password("raj").roles("USER");
}

@Override
protected void configure(HttpSecurity httpSecurity) throws Exception {
    httpSecurity
            .csrf().disable()
            .logout().disable()
            .formLogin().disable()
            .sessionManagement().sessionCreationPolicy(SessionCreationPolicy.STATELESS)
            .and()
                .anonymous()
            .and()
                .exceptionHandling().authenticationEntryPoint(
                        (req, rsp, e) -> rsp.sendError(HttpServletResponse.SC_UNAUTHORIZED))
            .and()
                .addFilterAfter(new JwtUsernamePasswordAuthenticationFilter(config, authenticationManager()),
                        UsernamePasswordAuthenticationFilter.class)
            .authorizeRequests()
                .antMatchers(config.getUrl()).permitAll()
                .anyRequest().authenticated();
}
```

Here I am using inMemoryAuthentication but we can use any authentication including that of JDBC to determine if the username and password are correct.

At the same time, we are also performing, JwtUsernamePasswordAuthenticationFilter filter that basically, generates a signed JWT token with all the claims, and places the same inside the body of the response to be used by the subsequent requests. I have used AbstractAuthenticationProcessingFilter for this purpose, which requires us to override 2 methods.

1. attemptAuthentication: The one which performs the actual Authentication. Remember a fact that, these username and password are compared against what we configured in the previous configure method.
2. successfullAuthentication.

```
public class JwtUsernamePasswordAuthenticationFilter extends AbstractAuthenticationProcessingFilter {

    private final JwtAuthenticationConfig config;
    private final ObjectMapper mapper;

    public JwtUsernamePasswordAuthenticationFilter(JwtAuthenticationConfig config, AuthenticationManager authManager)
        super(new AntPathRequestMatcher(config.getUrl(), "POST"));
        setAuthenticationManager(authManager);
        this.config = config;
        this.mapper = new ObjectMapper();
    }

    @Override
    public Authentication attemptAuthentication(HttpServletRequest req, HttpServletResponse rsp)
            throws AuthenticationException, IOException {
        User u = mapper.readValue(req.getInputStream(), User.class);
        return getAuthenticationManager().authenticate(new UsernamePasswordAuthenticationToken(
                u.getUsername(), u.getPassword(), Collections.emptyList()
```

```
        ));
    }

    @Override
    protected void successfulAuthentication(HttpServletRequest req, HttpServletResponse rsp, FilterChain chain,
                                            Authentication auth) {

        Instant now = Instant.now();
        String token = Jwts.builder()
                .setSubject(auth.getName())
                .claim("authorities", auth.getAuthorities().stream()
                        .map(GrantedAuthority::getAuthority).collect(Collectors.toList()))
                .setIssuedAt(Date.from(now))
                .setExpiration(Date.from(now.plusSeconds(config.getExpiration())))
                .signWith(SignatureAlgorithm.HS256, config.getSecret().getBytes())
                .compact();
        rsp.addHeader(config.getHeader(), config.getPrefix() + " " + token);
    }
```

backend-service

This is a simple SpringBootapplication with a restcontroller, that returns JSON responses.

Now we can perform the following requests in the same sequence. Remember, we are running zuul gateway service at 8080 and that will be used by any front end application like angular.

```
curl -i -H "Content-Type: application/json" -X POST -d
'{"username":"raj","password":"raj"}' http://localhost:8080/login
```

This would return a response with JWT token that can be used further like below:

```
curl -i -H "Authorization: Bearer token-recieved-above"
http://localhost:8080/backend/user
```

In the above application, we have seen how the zuul gateway is basically performing the authentication and then passing the requests to the backend. The same zuul gateway application can be configured to register to Eureka discovery service. That way we can configure Ribbon to talk to just zuul Gateway to perform the loadbalancing as well. I wish to publish more on my findings. Stay tuned.

Microservices Zuul