

JSON Web Token

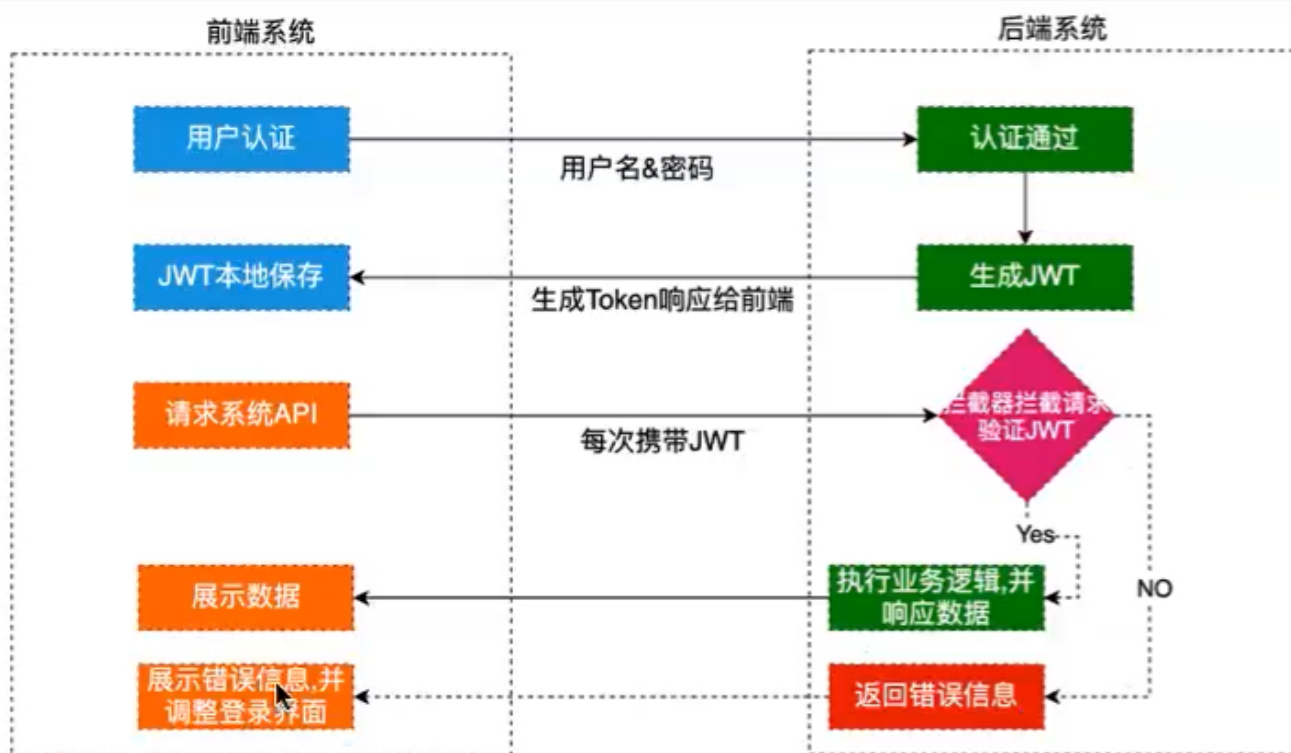
1. 什么是JWT?

- JSON Web Token (JWT)是一个开放标准(RFC 7519)，它定义了一种紧凑的、自包含的方式，用于作为JSON对象在各方之间安全地传输信息。该信息可以被验证和信任，因为它是数字签名的。

1.1 什么时候应该用JWT?

- Authorization (授权)**: 这是使用JWT的最常见场景。一旦用户登录，后续每个请求都将包含JWT，允许用户访问该令牌允许的路由、服务和资源。**单点登录**是现在广泛使用的JWT的一个特性，因为它的开销很小，并且可以轻松地跨域使用。
- Information Exchange (信息交换)**: 对于安全的在各方之间传输信息而言，JSON Web Tokens无疑是一种很好的方式。因为JWT可以被签名，例如，用公钥/私钥对，你可以确定发送人就是它们所说的那个人。另外，由于签名是使用头和有效负载计算的，您还可以验证内容没有被篡改。

1.2 认证流程



- 首先，前端通过Web表单将自己的用户名和密码发送到后端的接口。这一过程一般是一个HTTP **POST请求**。建议的方式是通过SSL加密的传输(https协议)，从而避免敏感信息被嗅探。

2. 后端核对用户名和密码成功后，将**用户的id等其他信息作为JWT Payload (负载)**，将其与头部分别进行Base64编码拼接后签名，形成一个JWT(Token)。形成的JWT就是一个形同11. zzz. xxx的字符串。**token head.payload.signature**
3. 后端将JWT字符串作为登录成功的返回结果**返回给前端**。前端可以将返回的结果保存在localStorage或sessionStorage上，退出登录时前端删除保存的JWT即可。
4. 前端在每次请求时将JWT**放入HTTP Header中的Authorization位**。(解决XSS和XSRF问题)
5. 后端检查是否存在，如存在验证JWT的有效性。
 - 检查签名是否正确;
 - 检查Token是否过期;
 - 检查Token的接收方是否是自己(可选)
6. 验证通过后后端使用JWT中包含的用户信息进行其他逻辑操作，返回相应结果。

1.3 JWT优势在哪？

1. 简洁(Compact): 可以通过URL，POST参数或者在HTTP header发送，数据量小，传输速度快
2. 自包含(Self-contained):**负载中包含了所有用户所需要的信息**，避免了多次查询数据库
3. 因为Token是 以**JSON加密的形式保存在客户端**的，所以JWT是跨语言的，原则上任何web形式都支持。
4. 不需要在服务端保存会话信息，特别**适用于分布式微服务**。

1.4 JWT具体包含信息

1. header

标头通常由两部分组成: **令牌类型**(即JWT) 和**所使用的签名算法**，例如HMAC、SHA256或RSA。它会使用Base64 编码组成JWT 结构的第一部分。

注意:Base64是一种编码，也就是说，它是可以被翻译回原来的样子来的。它并不是一种加密过程。

Java

```
1  {  
2      "alg": "HS256",  
3      "typ": "JWT"  
4  }
```

2. Payload

令牌的第二部分是有效负载，其中包含声明。**声明是有关实体(通常是用户)和其他数据的声明**。同样的，它会使用Base64 编码组成JWT结构的第二部分

JSON

```
1 {  
2     "sub" : "HS256"  
3     "name" : "yjiewei"  
4     "admin" : "true"  
5 }
```

3. Signature

header和payload都是结果Base64编码过的，中间用.隔开，第三部分就是前面两部分合起来做签名，密钥绝对自己保管好，签名值同样做Base64编码拼接在JWT后面。（签名并编码）

Apache

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

2. 光说不练假把式

2.1 整合pom

HTML

```
1 <dependencies>  
2     <dependency>  
3         <groupId>org.springframework.boot</groupId>  
4         <artifactId>spring-boot-starter-web</artifactId>  
5     </dependency>  
6  
7     <!--引入mybatis-->  
8     <dependency>  
9         <groupId>org.mybatis.spring.boot</groupId>  
10        <artifactId>mybatis-spring-boot-starter</artifactId>  
11        <version>2.1.0</version>  
12    </dependency>  
13  
14    <!--引入jwt-->  
15    <dependency>  
16        <groupId>com.auth0</groupId>  
17        <artifactId>java-jwt</artifactId>  
18        <version>3.10.3</version>  
19    </dependency>
```

```

20
21 <!--$/\mysql-->
22 <dependency>
23     <groupId>mysql</groupId>
24     <artifactId>mysql-connector-java</artifactId>
25     <version>8.0.16</version>
26 </dependency>
27
28 <!--$/\druid-->
29 <dependency>
30     <groupId>com.alibaba</groupId>
31     <artifactId>druid</artifactId>
32     <version>1.2.0</version>
33 </dependency>
34
35 <dependency>
36     <groupId>org.projectlombok</groupId>
37     <artifactId>lombok</artifactId>
38     <optional>true</optional>
39 </dependency>
40 <dependency>
41     <groupId>org.springframework.boot</groupId>
42     <artifactId>spring-boot-starter-test</artifactId>
43     <scope>test</scope>
44 </dependency>
45 </dependencies>

```

2.2 测试JWT加密过程

TypeScript

```

1  public class Jwt {
2
3      /**
4       * 获取JWT令牌
5       */
6      @Test
7      public void getToken() {
8          Map<String, Object> map = new HashMap<>();
9
10         Calendar instance = Calendar.getInstance();
11         instance.add(Calendar.SECOND, 2000);
12
13         /...

```

```

13      /**
14       * header可以不写有默认值
15       * payload 通常用来存放用户信息
16       * signature 是前两个合起来的签名值
17       */
18       String token = JWT.create().withHeader(map) //header
19           .withClaim("userId", 21) //payload
20           .withClaim("username", "yjiewei") //payload
21           .withExpiresAt(instance.getTime()) //指定令牌的过期时间
22           .sign(Algorithm.HMAC256("!RH04$%^fi$R")); //签名, 密钥自己记住
23       System.out.println(token);
24   }
25
26   /**
27    * 令牌验证:根据令牌和签名解析数据
28    * 常见异常:
29    *   SignatureVerificationException 签名不一致异常
30    *   TokenExpiredException         令牌过期异常
31    *   AlgorithmMismatchException     算法不匹配异常
32    *   InvalidClaimException          失效的payload异常
33    */
34   @Test
35   public void tokenVerify() {
36       // token值传入做验证
37       String token =
38           "eyJ0eXAiOiJKV1QiLCJhbGciOiJIUzI1NiJ9.eyJleHAiOiJlE2MjkyMDg0NjgsInVzZXJJZCI6MjEsInVzZXJuYVl1IjoieWppZXdlIiwiaS99.e4auZWkykZ2Hu8Q20toaks-4e62gerPLDEPHvhunCnQ";
39
40       /**
41        *  userId: 21
42        *  用户名: yjiewei
43        *  过期时间: Tue Aug 17 21:54:28 CST 2021
44        */
45       JWTVerifier jwtVerifier =
46           JWT.require(Algorithm.HMAC256("!RH04$%^fi$R")).build();
47       DecodedJWT decodedJWT = jwtVerifier.verify(token); // 验证并获取解码后的
48       token
49
50       System.out.println("userId: " + decodedJWT.getClaim("userId").asInt());
51       System.out.println("用户名: " +
52           decodedJWT.getClaim("username").asString());
53       System.out.println("过期时间: " + decodedJWT.getExpiresAt());
54   }
55 }

```

3. 封装工具类

Java就是封装抽象封装抽象...所以这里也封装一个工具类。

TypeScript

```
1  public class JWTUtil {
2
3      /**
4       * 密钥要自己保管好
5       */
6      private static String SECRET = "privatekey#^&^%!save";
7
8      /**
9       * 传入payload信息获取token
10      * @param map payload
11      * @return token
12      */
13     public static String getToken(Map<String, String> map) {
14         JWTCreator.Builder builder = JWT.create();
15
16         //payload
17         map.forEach(builder::withClaim);
18
19         Calendar instance = Calendar.getInstance();
20         instance.add(Calendar.DATE, 3); //默认3天过期
21
22         builder.withExpiresAt(instance.getTime()); //指定令牌的过期时间
23         return builder.sign(Algorithm.HMAC256(SECRET));
24     }
25
26     /**
27      * 验证token
28      */
29     public static DecodedJWT verify(String token) {
30         //如果有任何验证异常，此处都会抛出异常
31         return JWT.require(Algorithm.HMAC256(SECRET)).build().verify(token);
32     }
33
34     /**
35      * 获取token中的payload
36      */
37     public static Map<String, Claim> getPayloadFromToken(String token) {
38         return
39         JWT.require(Algorithm.HMAC256(SECRET)).build().verify(token).getClaims();
40     }
```

4. JWT 整合SpringBoot

我们要对每个请求都去验证实单点登录，通过拦截器拦截请求对JWT做验证。

4.1 登录并获取token

GET

http://localhost:8001/user/login?name=yjiewei&password=123

Params

Send

Save

Key	Value	Description	...	Bulk Edit
<input checked="" type="checkbox"/> name	yjiewei			
<input checked="" type="checkbox"/> password	123			
New key	Value	Description		

AuthorizationHeadersBodyPre-request ScriptTestsCookiesCode

Key	Value	Description	...	Bulk Edit	Presets
New key	Value	Description			

BodyCookies (1)Headers (5)Test ResultsStatus: 200 OKTime: 5616 msSize: 374 B

PrettyRawPreviewJSON

```
1 {
2   "msg": "登录成功",
3   "status": true,
4   "token": "eyJ0eXAiOiJKV1QiLCJhbGciOiJIUzI1NiJ9
    .eyJwYXNkd29yZCI6IjEyMyIsIm5hbWUiOiJ5amld2VpIiwiaWQiOiJxIiwiaXhwIjoxNjI5NDY1MjEwQ000LX_1Vxxt-DelP1my2BtWk
    -nmIG9q7W1m_QMiuS26l8"
5 }
```

4.2 在请求头中添加token

The screenshot shows the Postman interface for a POST request to `http://localhost:8001/user/test`. The 'Headers' tab is active, displaying a table with one header: 'token' with a value `eyJ0eXAiOiJKV1QiLCJhbGciOiJIUzI1NiJ9.eyJwYXNzd2...`. The 'Body' tab is also visible, showing a JSON response: `{ "msg": "请求成功", "status": true }`. The status bar at the bottom indicates a 200 OK response.

Key	Value	Description
<input checked="" type="checkbox"/> token	eyJ0eXAiOiJKV1QiLCJhbGciOiJIUzI1NiJ9.eyJwYXNzd2...	

Key	Value	Description
New key	Value	Description


```

1 {
2   "msg": "请求成功",
3   "status": true
4 }
  
```

Status: 200 OK Time: 136 ms

4.3 重要的代码片段

TypeScript

```
1  // 拦截器
2  @Slf4j
3  public class JWTInterceptor implements HandlerInterceptor {
4
5      @Override
6      public boolean preHandle(HttpServletRequest request,
7                              HttpServletResponse response,
8                              Object handler) throws Exception {
9
10         //获取请求头中的令牌
11         String token = request.getHeader("token");
12         log.info("当前token为: {}", token);
13
14         Map<String, Object> map = new HashMap<>();
15         try {
16             JWTUtil.verify(token);
17             return true;
18         } catch (SignatureVerificationException e) {
19             e.printStackTrace();
20             map.put("msg", "签名不一致");
21         } catch (TokenExpiredException e) {
22             e.printStackTrace();
23             map.put("msg", "令牌过期");
24         } catch (AlgorithmMismatchException e) {
25             e.printStackTrace();
26             map.put("msg", "算法不匹配");
27         } catch (InvalidClaimException e) {
28             e.printStackTrace();
29             map.put("msg", "失效的payload");
30         } catch (Exception e) {
31             e.printStackTrace();
32             map.put("msg", "token无效");
33         }
34
35         map.put("status", false);
36
37         //响应到前台: 将map转为json
38         String json = new ObjectMapper().writeValueAsString(map);
39         response.setContentType("application/json;charset=UTF-8");
40         response.getWriter().println(json);
41         return false;
42     }
```

```
43 }
```

Java

```
1 // 指定拦截路径
2 @Configuration
3 public class InterceptorConfig implements WebMvcConfigurer {
4
5     @Override
6     public void addInterceptors(InterceptorRegistry registry) {
7         registry.addInterceptor(new JWTInterceptor())
8             .addPathPatterns("/user/test")
9             .excludePathPatterns("/user/login");
10    }
11 }
```

TypeScript

```
1 // 拦截请求并验证请求头中的token
2 @PostMapping("/user/test")
3 public Map<String, Object> test(HttpServletRequest request) {
4     String token = request.getHeader("token");
5     DecodedJWT verify = JWTUtil.verify(token);
6     String id = verify.getClaim("id").asString(); // 我前面存的时候转字符串了
7     String name = verify.getClaim("name").asString();
8     log.info("用户id: {}", id);
9     log.info("用户名: {}", name);
10
11     //TODO:业务逻辑
12     Map<String, Object> map = new HashMap<>();
13     map.put("status", true);
14     map.put("msg", "请求成功");
15     return map;
16 }
```