# SpringBoot整合Token验证

# JWT

## 什么是JSON Web Token？

JSON Web Token（JWT）是一个开放标准（[RFC 7519](https://tools.ietf.org/html/rfc7519)），它定义了一种紧凑且独立的方式，用于在各方之间作为JSON对象安全地传输信息。此信息可以通过数字签名进行验证和信任。JWT可以使用秘密（使用**HMAC**算法）或使用**RSA**或**ECDSA**的公钥/私钥对进行**签名**。

虽然JWT可以加密以在各方之间提供保密，但我们将专注于*签名*令牌。签名令牌可以验证其中包含的声明的*完整性*，而加密令牌则*隐藏*其他方的声明。当使用公钥/私钥对签名令牌时，签名还证明只有持有私钥的一方是签署它的一方。

## 什么时候应该使用JSON Web令牌？

以下是JSON Web令牌有用的一些场景：

* **授权**：这是使用JWT的最常见方案。一旦用户登录，每个后续请求将包括JWT，允许用户访问该令牌允许的路由，服务和资源。Single Sign On是一种现在广泛使用JWT的功能，因为它的开销很小，并且能够在不同的域中轻松使用。
* **信息交换**：JSON Web令牌是在各方之间安全传输信息的好方法。因为JWT可以签名 - 例如，使用公钥/私钥对 - 您可以确定发件人是他们所说的人。此外，由于使用标头和有效负载计算签名，您还可以验证内容是否未被篡改。

## 什么是JSON Web令牌结构？

在紧凑的形式中，JSON Web Tokens由dot（.）分隔的三个部分组成，它们是：

* Header
* Payload
* Signature

因此，JWT通常如下所示。

xxxxx.yyyyy.zzzzz

让我们分解不同的部分。

### Header

标头*通常*由两部分组成：令牌的类型，即JWT，以及正在使用的签名算法，例如HMAC SHA256或RSA。

例如：

{  
 "alg": "HS256",  
 "typ": "JWT"  
}

然后，这个JSON被编码为**Base64Url**，形成JWT的第一部分。

### Payload

令牌的第二部分是payload，其中包含声明。声明是关于实体（通常是用户）和其他数据的声明。声明有三种类型：*注册*，*公开*和*私有声明*。

* **[已注册的声明](https://tools.ietf.org/html/rfc7519#section-4.1)**：这些是一组预定义声明，不是强制性的，但建议使用，以提供一组有用的，可互操作的声明。其中一些是：**iss** (issuer), **exp** (expiration time), **sub** (subject), **aud**(audience)等。
* 请注意，声明名称只有三个字符，因为JWT意味着紧凑。
* **[公开声明](https://tools.ietf.org/html/rfc7519#section-4.2)**：这些可以由使用JWT的人随意定义。但为避免冲突，应在[IANA JSON Web令牌注册表](https://www.iana.org/assignments/jwt/jwt.xhtml)中定义它们，或者将其定义为包含防冲突命名空间的URI。
* **[私有声明](https://tools.ietf.org/html/rfc7519#section-4.3)**：私有声明是提供者和消费者所共同定义的声明，一般不建议存放敏感信息，因为base64是对称解密的，意味着该部分信息可以归类为明文信息。

示例payload可以是：

{  
 "sub": "1234567890",  
 "name": "John Doe",  
 "admin": true  
}

然后，payload经过**Base64Url**编码，形成JSON Web令牌的第二部分。

请注意，对于签名令牌，此信息虽然可以防止被篡改，但任何人都可以读取。除非加密，否则不要将秘密信息放在JWT的payload或header中。

### Signature

要创建签名部分，必须采用 header, payload, secret，标头中指定的算法，并对其进行签名。

例如，如果要使用HMAC SHA256算法，将按以下方式创建签名：

HMACSHA256(  
 base64UrlEncode(header) + "." +  
 base64UrlEncode(payload),  
 secret)

签名用于验证消息在此过程中未被更改，并且，在使用私钥签名的令牌的情况下，它还可以验证JWT的发件人是否是它所声称的人。

### 全部放在一起

输出是三个由点分隔的Base64-URL字符串，可以在HTML和HTTP环境中轻松传递，而与基于XML的标准（如SAML）相比更加紧凑。

下面显示了一个JWT，它具有先前的头和有效负载编码，并使用机密签名。 

图1 签名图

## JWT工作原理

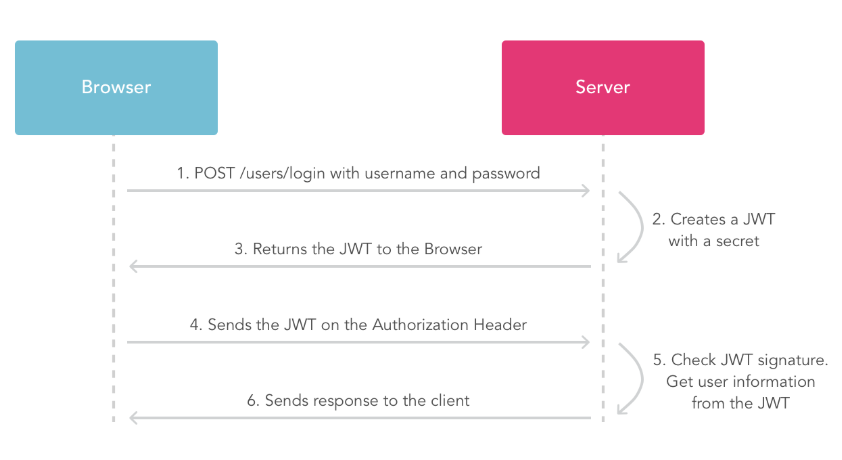


图2 工作原理图

1）用户登陆的时候使用用户名和密码发送POST请求。

2）服务器使用私钥创建一个jwt。

3）服务器返回这个jwt到浏览器。

4）浏览器将该jwt串加入请求头中向服务器发送请求。

5）服务器验证jwt。

6）返回相应的资源给客户端。

# SpringBoot集成

### 引入依赖

<dependency>  
 <groupId>com.auth0</groupId>  
 <artifactId>java-jwt</artifactId>  
 <version>3.4.0</version>  
</dependency>

### 定义注解

**PassToken：**跳过验证。

@Target({ElementType.METHOD, ElementType.TYPE})  
@Retention(RetentionPolicy.RUNTIME)  
public @interface PassToken {  
 boolean required() default true;  
}

UserLoginToken：需要验证。

@Target({ElementType.METHOD, ElementType.TYPE})  
@Retention(RetentionPolicy.RUNTIME)  
public @interface UserLoginToken {  
 boolean required() default true;  
}

### 定义实体类

public class UserDo {  
 private String id;  
  
 private String userName;  
  
 private String userPasswd;  
  
 public String getId() {  
 return id;  
 }  
  
 public void setId(String id) {  
 this.id = id == null ? null : id.trim();  
 }  
  
 public String getUserName() {  
 return userName;  
 }  
  
 public void setUserName(String userName) {  
 this.userName = userName == null ? null : userName.trim();  
 }  
  
 public String getUserPasswd() {  
 return userPasswd;  
 }  
  
 public void setUserPasswd(String userPasswd) {  
 this.userPasswd = userPasswd == null ? null : userPasswd.trim();  
 }  
}

### 编写TokenServiceImpl

@Service  
public class TokenServiceImpl implements TokenService {  
 @Override  
 public String getToken(UserDo userDo) {  
 String token="";  
 token= JWT.create().withAudience(userDo.getId())  
 .sign(Algorithm.HMAC256(userDo.getUserPasswd()));  
 return token;  
 }  
}

### 编写拦截器

public class AuthenticationInterceptor implements HandlerInterceptor {  
 @Reference  
 UserService userService;  
  
 @Override  
 public boolean preHandle(HttpServletRequest httpServletRequest, HttpServletResponse httpServletResponse, Object object) throws Exception {  
 String token = httpServletRequest.getHeader("token");// 从 http 请求头中取出 token  
 // 如果不是映射到方法直接通过  
 if (!(object instanceof HandlerMethod)) {  
 return true;  
 }  
 HandlerMethod handlerMethod = (HandlerMethod) object;  
 Method method = handlerMethod.getMethod();  
 //检查是否有passtoken注释，有则跳过认证  
 if (method.isAnnotationPresent(PassToken.class)) {  
 PassToken passToken = method.getAnnotation(PassToken.class);  
 if (passToken.required()) {  
 return true;  
 }  
 }  
 //检查有没有需要用户权限的注解  
 if (method.isAnnotationPresent(UserLoginToken.class)) {  
 UserLoginToken userLoginToken = method.getAnnotation(UserLoginToken.class);  
 if (userLoginToken.required()) {  
 // 执行认证  
 if (token == null) {  
 throw new RuntimeException("无token，请重新登录");  
 }  
 // 获取 token 中的 user id  
 String userId;  
 try {  
 userId = JWT.decode(token).getAudience().get(0);  
 } catch (JWTDecodeException j) {  
 throw new RuntimeException("401");  
 }  
 UserDo user = userService.findUserById(userId);  
 if (user == null) {  
 throw new RuntimeException("用户不存在，请重新登录");  
 }  
 // 验证 token  
 JWTVerifier jwtVerifier = JWT.require(Algorithm.HMAC256(user.getUserPasswd())).build();  
 try {  
 jwtVerifier.verify(token);  
 } catch (JWTVerificationException e) {  
 throw new RuntimeException("401");  
 }  
 return true;  
 }  
 }  
 return true;  
 }  
  
 @Override  
 public void postHandle(HttpServletRequest httpServletRequest,  
 HttpServletResponse httpServletResponse,  
 Object o, ModelAndView modelAndView) throws Exception {  
  
 }  
  
 @Override  
 public void afterCompletion(HttpServletRequest httpServletRequest,  
 HttpServletResponse httpServletResponse,  
 Object o, Exception e) throws Exception {  
 }  
}

流程图如下：

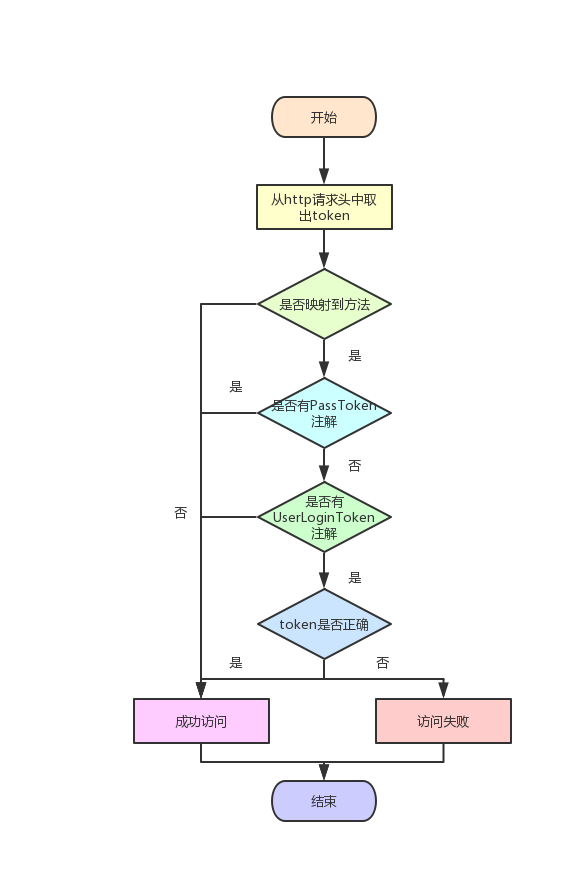


图3 拦截流程图

### 配置拦截器

@Configuration  
public class InterceptorConfig implements WebMvcConfigurer {  
 @Override  
 public void addInterceptors(InterceptorRegistry registry) {  
 registry.addInterceptor(authenticationInterceptor())  
 .addPathPatterns("/\*\*");  
 }  
 @Bean  
 public AuthenticationInterceptor authenticationInterceptor() {  
 return new AuthenticationInterceptor();  
 }  
}

### controller-api

@RestController  
@CrossOrigin  
@RequestMapping("/user")  
@Api(tags = "用户登录")  
public class LoginController extends BaseController {  
 @Reference  
 UserService userService;  
 @Reference  
 TokenService tokenService;  
  
 //登录  
 @PostMapping("/login")  
 @ResponseBody  
 public Object login(UserDo user) {  
 JSONObject jsonObject = new JSONObject();  
 UserDo userForBase = userService.findUserByUsername(user.getUserName());  
 if (userForBase == null) {  
 jsonObject.put("message", "登录失败,用户不存在");  
 return jsonObject;  
 } else {  
 if (!userForBase.getUserPasswd().equals(user.getUserPasswd())) {  
 jsonObject.put("message", "登录失败,密码错误");  
 return jsonObject;  
 } else {  
 String token = tokenService.getToken(userForBase);  
 jsonObject.put("token", token);  
 jsonObject.put("user", userForBase);  
 return jsonObject;  
 }  
 }  
 }  
  
 @UserLoginToken  
 @GetMapping("/getMessage")  
 public String getMessage() {  
 return "你已通过验证";  
 }  
}

即传入token才可以调用getMessage（）方法。

# 接口调试

### 调用getMessage()

使用postmen调用接口：http://localhost:8094/user/getMessage。

输出错误信息如下：

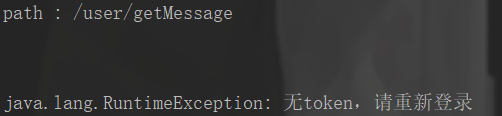


图4 接口调试图

### 调用login()

使用postmen调用接口：http://localhost:8094/user/login

传入id,用户名和密码三个参数，输出结果如下：

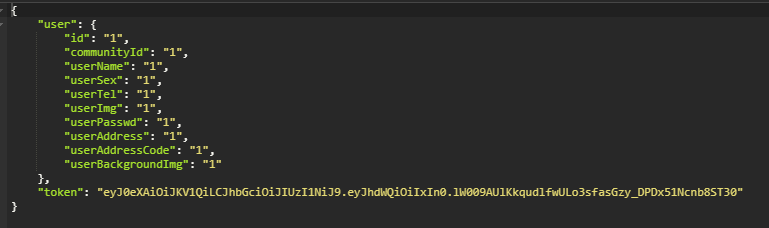


图5 接口调试图

### 再次调用getMessage()

在Headers中加入token参数，输出结果如下：

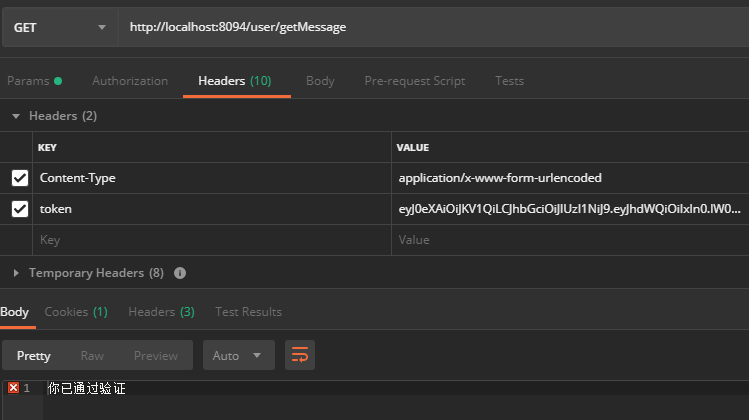


图6 接口调试图