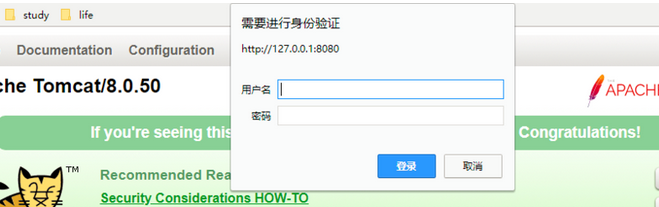
# 1. Http Basic Auth

这是一种最古老的安全认证方式，这种方式就是简单的访问API的时候，带上访问的username和password，由于信息会暴露出去，所以现在也越来越少用了，现在都用更加安全保密的认证方式，可能某些老的平台还在用。

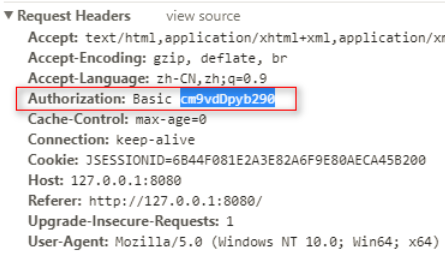
如下图所示，弹出一个框，让你填写用户名密码。这就是Tomcat自带的HTTPBasic认证。



当用户名密码输入错误后，会返回401 Unauthorized 表明认证失败，无法访问应用。

当认证成功后，你再看看详细的request headers，会发现多了一个请求头：

Authorization: Basic xxxXXXxxx



这就是你访问应用的凭据了，那段xxxXXX字符串是我写的表示这是一段密文，这是一段什么密文呢，就是讲用户名和密码进行一个Base64加密后得到的密文。所以你现在是不是也有同感了---这tm也太容易盗取了，所以现在新的应用几乎不怎么用这种方式了，虽然简单，但是安全级别太低了。

# 2. OAuth2

本人之前的博客介绍过OAuth2 以及使用Azure AD实现OAuth2认证方式，在这里呢，还是把那篇博客部分内容抠出来，方便大家总结查看。

<https://blog.csdn.net/aHardDreamer/article/details/88650939>

OAuth 即：Open Authrization(开放授权)， 它是一个开放标准，允许用户让第三方应用访问该用户在某一网站上存储的私密资源，而无需将用户名和密码提供给第三方。比如我们熟知的通过qq/微信/微博等登录第三方平台。OAuth 1.0版本发布后有许多安全漏洞，所以在OAuth2.0里面完全废止了OAuth1.0，它关注客户端开发者的简易性，要么通过组织在资源拥有者和HTTP服务商之间的被批准的交互动作代表用户，要么允许第三方应用代表用户获得访问的权限。读起来有点绕口，其实原理也非常简单，请看下面讲解。

一、首先我们要了解在OAuth2 认证和授权的过程中有这三个角色：

1. 服务提供方：顾名思义，提供受保护的服务和资源的，用户在这里面存了很多东西。

2. 用户： 存了东西（照片，资料等）在服务提供方的人。

3. 客户端：服务调用方，它要访问服务提供方的资源，需要在服务提供方进行注册，不然服务提供方不鸟它呀。

二、OAuth2 认证和授权的过程：

1）用户想操作存放在服务提供方的资源；

2）用户登录客户端，客户端向服务提供方请求一个临时token;

3）服务提供方验证客户端的身份后，给它一个临时token;

4）客户端获得临时token之后，将用户引导至服务提供方的授权页面，并请求用户授权。（在这个过程中会将临时token和客户端的回调链接/接口 发送给服务提供方 ---很明显服务提供方到时会回来call这个接口在用户认证并授权之后）

5）用户输入用户名密码登录，登录成功之后，可以授权客户端访问服务提供方的资源；

6）授权成功后，服务提供方将用户引导至客户端的网页（call第4步里面的回调链接/接口）；

7）客户端根据临时token从服务提供方那里获取正式的access token；

8）服务提供方根据临时token以及用户的授权情况授予客户端access token;

9）客户端使用access token访问用户存放在服务提供方的受保护的资源。



三、拿access token的方法（Grant Type）有下面四种，每一种都有适用的应用场景：

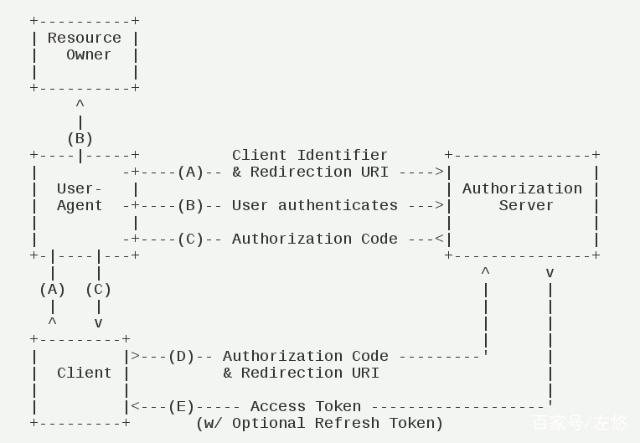
1. Authorization Code （授权码模式）

结合普通服务器端应用使用。

1）用户访问客户端，后者将前者导向认证服务器，假设用户给予授权，认证服务器将用户导向客户端事先指定的"重定向URI"（redirection URI），同时附上一个授权码。

2)客户端收到授权码，附上早先的"重定向URI"，向认证服务器申请令牌：GET /oauth/token?response\_type=code&client\_id=test&redirect\_uri=重定向页面链接。请求成功返回code授权码，一般有效时间是10分钟。

3）认证服务器核对了授权码和重定向URI，确认无误后，向客户端发送访问令牌（access token）和更新令牌（refresh token）。POST /oauth/token?response\_type=authorization\_code&code=SplxlOBeZQQYbYS6WxSbIA&redirect\_uri=重定向页面链接。请求成功返回access Token和refresh Token。



2. Implicit（简化模式）

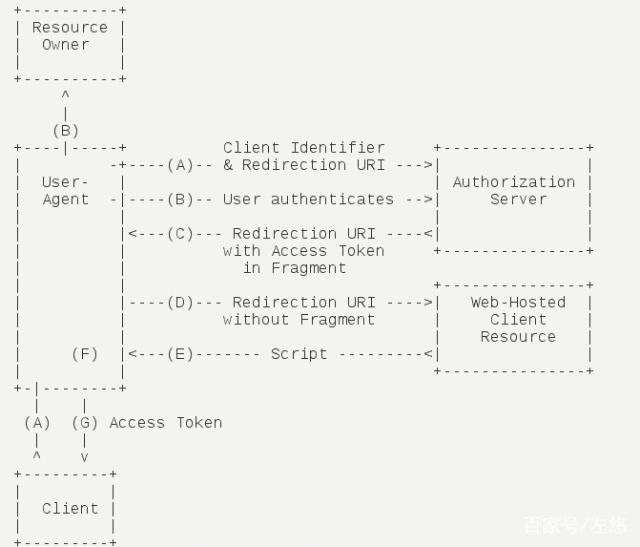
结合移动应用或 Web App 使用。

Access Token直接从授权服务器返回(只有前端渠道)

不支持refresh tokens

假定资源所有者和公开客户应用在同一个设备上

最容易受安全攻击



3. Resource Owner Password Credentials

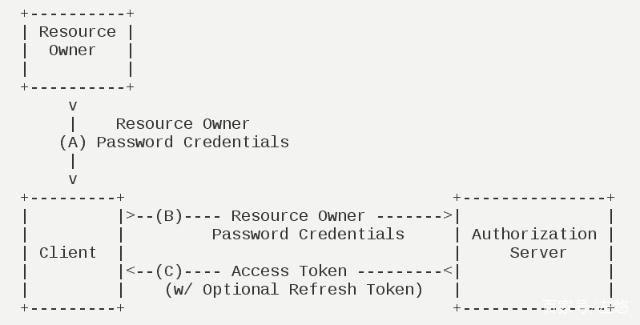
适用于受信任客户端应用，例如同个组织的内部或外部应用。

使用用户名密码登录的应用，例如桌面App

使用用户名/密码作为授权方式从授权服务器上获取access token

一般不支持refresh token

假定资源拥有者和公开客户在相同设备上

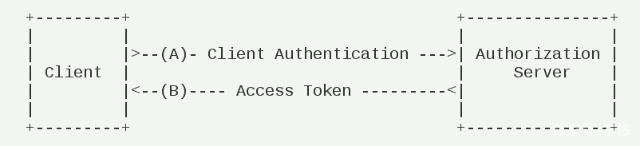


4. Client Credentials

适用于客户端调用主服务API型应用（比如百度API Store，不同项目之间的微服务互相调用）

只有后端渠道，使用客户凭证获取一个access token

因为客户凭证可以使用对称或者非对称加密，该方式支持共享密码或者证书



# 3. Cookie-Session Auth

Cookie-Session 认证机制在我们初学J2EE的时候接触的比较多，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；

但是这种基于cookie-session的认证使应用本身很难得到扩展，随着不同客户端用户的增加，独立的服务器已无法承载更多的用户，而这时候基于session认证应用的问题就会暴露出来。

**基于session认证所显露的问题：**

1）Session 增多会增加服务器开销

每个用户经过我们的应用认证之后，我们的应用都要在服务端做一次记录，以方便用户下次请求的鉴别，通常而言session都是保存在内存中，而随着认证用户的增多，服务端的开销会明显增大。

2）分布式或多服务器环境中适应性不好

用户认证之后，服务端做认证记录，如果认证的记录被保存在内存中的话，这意味着用户下次请求还必须要请求在这台服务器上,这样才能拿到授权的资源，这样在分布式的应用上，相应的限制了负载均衡器的能力。这也意味着限制了应用的扩展能力。不过，现在某些服务器可以通过设置粘性Session，来做到每台服务器之间的Session共享。

3）容易遭到CSRF攻击

因为是基于cookie来进行用户识别的, cookie如果被截获，用户就会很容易受到跨站请求伪造的攻击

# 4. Token Auth

基于token的鉴权机制类似于http协议也是无状态的，它不需要在服务端去保留用户的认证信息或者会话信息。这就意味着基于token认证机制的应用不需要去考虑用户在哪一台服务器登录了，这就为应用的扩展提供了便利。

流程：

用户使用用户名密码来请求服务器

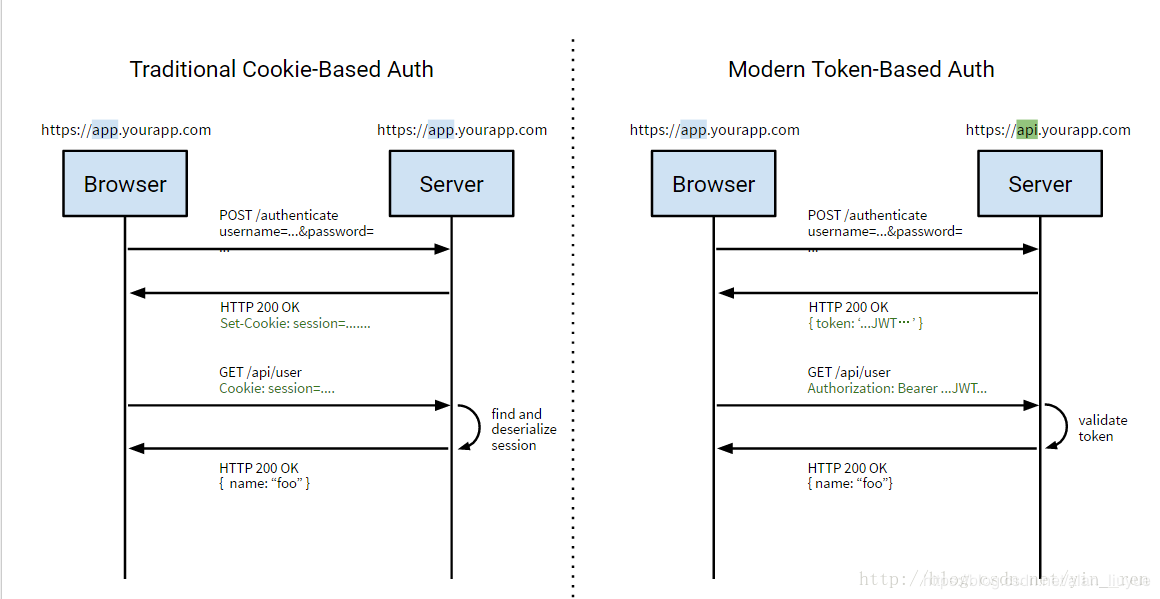
服务器进行验证用户的信息

服务器通过验证发送给用户一个token

客户端存储token，并在每次请求时附送上这个token值

服务端验证token值，并返回数据

这个token必须要在每次请求时传递给服务端，它应该保存在请求头里， 另外，服务端要支持CORS(跨来源资源共享)策略，一般我们在服务端这么做就可以了Access-Control-Allow-Origin。



# 5. JWT 认证机制（Json Web Token）

JWT作为一个开放的标准（RFC 7519），定义了一种**简洁的，自包含**的方法**用于通信双方之间以Json对象的形式安全的传递信息**。因为**数字签名**的存在，这些信息是可信的，JWT可以使用HMAC算法或者是RSA的公私秘钥对进行签名。

**简洁性**

可以通过URL，POST参数或者在HTTP header发送，因为数据量小，传输速度也很快

**自包含性**

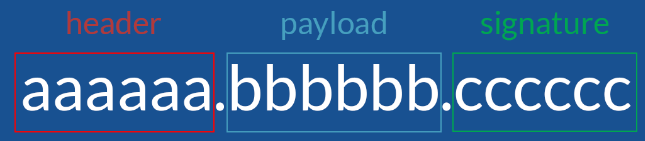
负载中包含了所有用户所需要的信息，避免了多次查询数据库

下列场景中使用JSON Web Token是很有用的：

**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JWTs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。

JWT的结构：

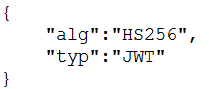


通过这张图，很清晰看出JWT的结构分为三部分，他们之间用“.”连接：

**Header：**

header典型的由两部分组成：token的类型（“JWT”）和算法名称（比如：HMAC SHA256或者RSA等等）。

例如：



然后，用Base64对这个JSON编码就得到JWT的第一部分

**Payload:**

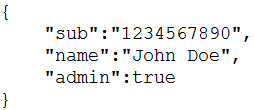
JWT的第二部分是payload，它包含声明（要求）。声明是关于实体(通常是用户)和其他数据的声明。声明有三种类型: registered, public 和 private。

Registered claims : 这里有一组预定义的声明，它们不是强制的，但是推荐。比如：iss (issuer), exp (expiration time), sub (subject), aud (audience)等。

Public claims : 可以随意定义。

Private claims : 用于在同意使用它们的各方之间共享信息，并且不是注册的或公开的声明。

下面是一个例子：



对payload进行Base64编码就得到JWT的第二部分

注意，不要在JWT的payload或header中放置敏感信息，除非它们是加密的。

**Signature:**

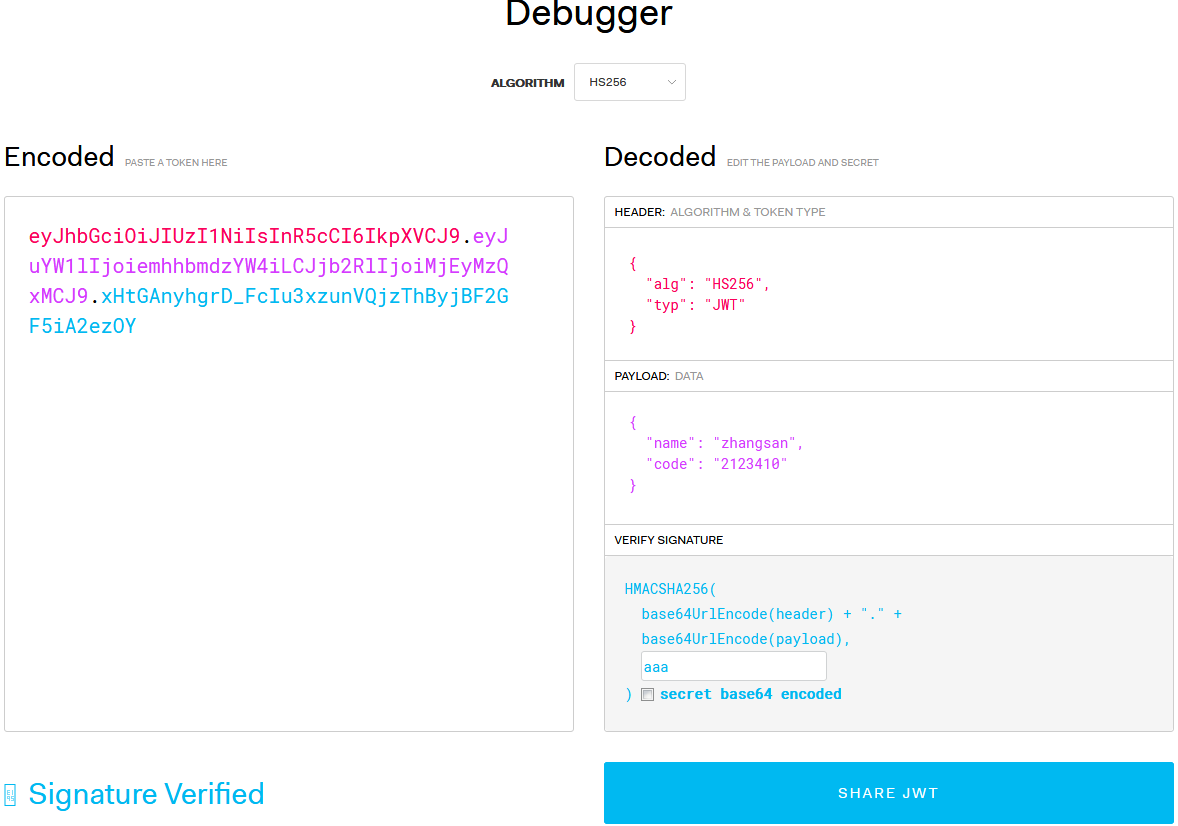
为了得到签名部分，你必须有编码过的header、编码过的payload、一个秘钥，签名算法是header中指定的那个，然对它们签名即可。

例如：

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

签名是用于验证消息在传递过程中有没有被更改，并且，对于使用私钥签名的token，它还可以验证JWT的发送方是否为它所称的发送方。

碰到JWT token可以去JWT官网解密看看，下面这是官网解密出来的数据，可以很清楚的看到它的三部分内容：



更多关于JWT的内容，可以前往这个博客：

<https://www.cnblogs.com/cjsblog/p/9277677.html>

参考：

<https://www.jianshu.com/p/f8c43dcd8b69>

<https://blog.csdn.net/alan_liuyue/article/details/88183267>

<https://www.cnblogs.com/cjsblog/p/9277677.html>