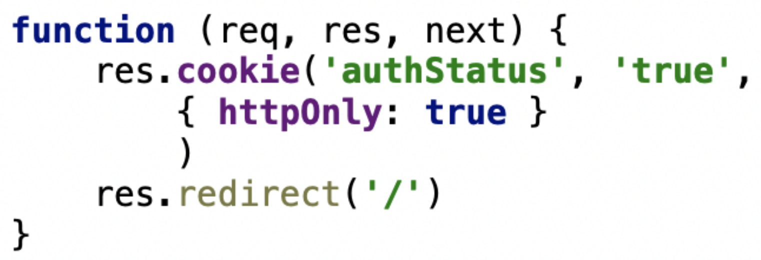
CAS CS 411
Lec 10

OAuth and JWTs

1. Sessions
    a. HTTP is stateless
    b. Nevertheless we usually want users to feel that they have established a stateful connection in the app
    c. Each language approaches this a little differently in terms of structure, but generally we'll do one of these:
        i. Use hidden fields in forms to pass data to the server in the query string
        ii. Pass a single userID to the server to use as a key into a database, i.e. on request body
        iii. Use a cookie to move data back and forth
        iv. For browser-side JavaScript (Angular, React, Vue, etc) use either session or local storage on the client (but still pass data back to the server)
        v. Pass data on HTTP headers
2. Stateless Servers
    a. Ironically we WANT the server endpoints to be stateless
    b. For the most part server-side transactions should pass ACID test
    c. If they weren't it might be difficult to scale the app with increasing load
    d. Note that there are stateful solutions such as web sockets if you need them
    e. In some cases we want to simply track the user through an app
        i. What page did they just see
        ii. What is their display name
        iii. What are their prefs for a given page
        iv. Probably fine to pass these values on the session or cookie
    f. In other cases we want to protect server / API endpoints
        i. Might be role-based
        ii. Could have logged in vs anonymous users
        iii. Here we tend to use authentication tokens passed back and forth
3. The Two auths - What's the Difference?
    a. There are two unfortunately similarly named concepts
    b. Authentication
        i. Create credentials asserting the identity of a user
    c. Authorization
        i. Grant access to resources (often via authentication)
    d. For our discussion we'll combine these and just talk about 'auth'

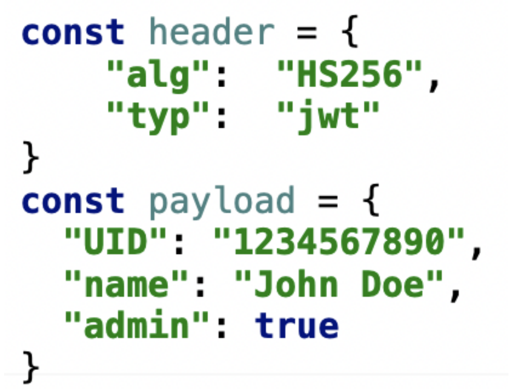
4. Simplest Case
    a. In the very simplest case we might just add a flag to a cookie to indicate that a user is logged in



```
function (req, res, next) {
    res.cookie('authStatus', 'true',
        { httpOnly: true }
        )
    res.redirect('/')
}
```

    b.
    c. The httpOnly property on the cookie means that the client can't read it…only the server (this limits its use on the client, though)
        i. Partly to limit access from JavaScript and mitigate XSS attacks
    d. The cookies stored on the client are automatically sent with each HTTP request
    e. One problem with this is that if we limit visibility with httpOnly we can't use data from the cookie on the client side
    f. If we remove the restriction anyone can copy the cookie and impersonate a user - the server wouldn't know the difference since cookies are sent essentially in clear text
    g. We need a way to tie a cookie to a user in an irrefutable way
5. JWTs
    a. One way is to cryptographically sign a cookie in a way that provides irrefutable identity
    b. This has been generalized into a technique called JSON Web Tokens (JWTs)
    c. Note that even though JSON looks JavaScripty, JWTs can (and are) used in just about every framework and language
    d. Each framework has packages available to work with JWTs
6. JWT Format
    a. A JSON Web Token has three parts
        i. A header that identifies the type of the token being generated and the signing algorithm
        ii. A payload that contains user-specific data such as a unique user ID to use on the server for lookups, creation / expiration timestamps, roles, and so on (the items in the payload are called claims)
        iii. A signature generated cryptographically from the header and payload, plus a secret value
        iv. The header and payload are each Base64 encoded; the signature is Base64 encoded after it is generated
        v. The three pieces are concatenated and delimited by a period (.)

        vi. Here’s an example (from jwt.io) of a plain-text JWT and the encoded, signed result:

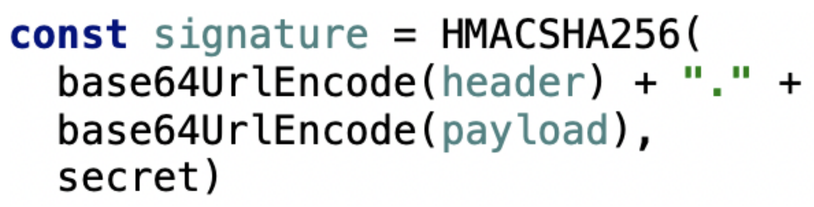


```
const header = {
    "alg":  "HS256",
    "typ":  "jwt"
}
const payload = {
  "UID": "1234567890",
  "name": "John Doe",
  "admin": true
}
```

7. Signing Algorithms - Hash
    a. In the previous example the signing algorithm is a hash (HMAC256)
    b. Hashes are
        i. irreversible
        ii. reproducible
        iii. collision-free
        iv. unpredictable
    c. This works well with one glaring problem (which we’ll see in a moment)
8. Passing the JWT
    a. The JWT can be passed via cookie, an HTTP Authorization header (usually as a bearer token), or placed on the session
    b. The cookie is the easiest since it is always transmitted with the HTTP request, although it is vulnerable to interception; setting httpOnly to true will help
    c. When using a header such as the

    ```
    res.header('Authorization: Bearer ‘ + jwt);
    ```

    header must be manually added on both the client and server side
    d. It isn’t uncommon to store the JWT in local storage on the client side to persist it between browser sessions (there’s also session storage on the client that is deleted on browser exit)
9. Validating a JWT
    a. Recall that the header and payload are sent in clear text (Base64 encoded)
    b. To validate a JWT by its signature, either on the client or server…
        i. Use the same hashing algorithm no the encoded header and payload
        ii. Must add the secret value
        iii. Signatures should be the same
    c. What is the glaring problem?
    d. Recall that the payload and header are combined with a secret value and then hashed to produce the signature



```
const signature = HMACSHA256(
  base64UrlEncode(header) + "." +
  base64UrlEncode(payload),
  secret)
```

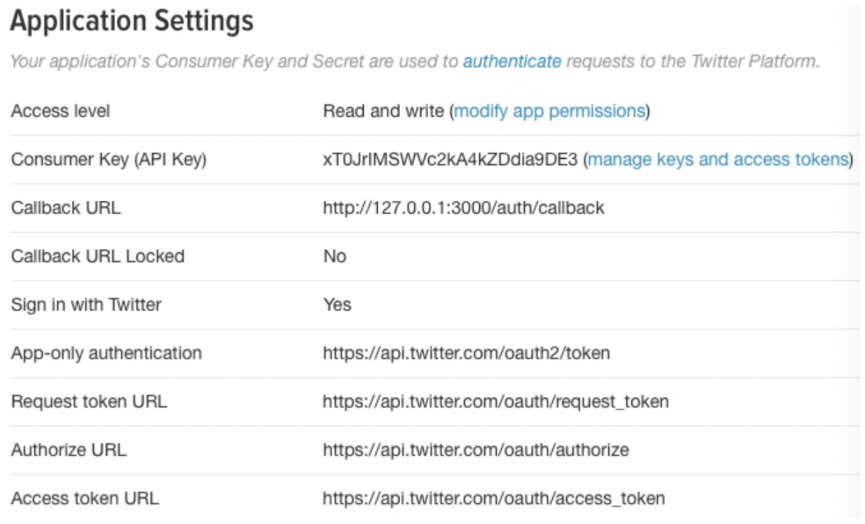
    e. For validation, both sides need to know the secret value
    f. Exposing the value on the client side is a serious risk

10. Using RSA to Sign JWTs
    a. Instead of using a shared secret (with its security risks and distribution pain) we can leverage RSA public/private key pairs for signing JWTs
    b. This protects the private key in one spot
    c. The public key can be faced with an endpoint for retrieval
    d. Signing is done with the private key on the server side
    e. Validation can be done everywhere with the public key
    f. Note we are NOT encrypting, just signing ▪ It might affect performance to encrypt and decrypt large payloads
        i. Instead a hash is taken on Base64 header and payload, and the hash is encrypted
        ii. The hash is a fixed length, and so the signing performance is independent of the size of the payload
        iii. To reiterate, the header and payload are still sent in the clear!
        iv. BTW this is how RSA/PGP email signatures work
11. Using the Token
    a. When the server has validated a user (via username / password, for example) it generates a JWT and returns it to the client
    b. The client and server use the JWT as an authorization token
    c. On the server side it is a quick operation to validate the signature and parse the payload
    d. If the JWT is cookie based, the client side doesn't need to do anything at all, since the cookie, and thus the authorization token, is automatically passed with each request
    e. The client can then access secured endpoints on the server
12. Third-Party Authentication
    a. What if we want to avoid the whole hassle of storing credentials such as passwords in our application?
    b. We can leverage protocols such as OAuth to hand authentication functions to a trusted third party
    c. The third party handles all of the authentication and returns a secure token that the app can use when working with a user

d. Fun fact: OAuth came out of Twitter as a protocol to consolidate some early social bookmarking tools with OpenID around 2006-2007
e. Another fun fact…OAuth really is intended for authorization, but is widely used for authentication
f. There are two versions of OAuth
   i. OAuth 1.0a (Spotify, Twitter for user auth)
   ii. OAuth 2.0 (Facebook, Twitter for app-only auth)
g. The versions differ in their complexity…2.0 was created to simplify what many felt was overkill in 1.0a
h. 2.0 is still in draft but many sites are switching to it…1.0a is a bit awkward for mobile apps
i. 1.0a also doesn't scale all that well
j. 1.0a requires a crypto signature on requests, using client ID and a shared secret…2.0 drops this and requires HTTPS for secure transport

13. Two Flavors of the Two Versions
   a. Both OAuth versions provide for authentication of a user
   b. Authorization, also available in both, allows for application to make requests as a proxy on the user's behalf
      i. This is set up with a grant type list during authorization
      ii. It's why you see a list on the popup on the client side….this app would like to read your timeline, post tweets on your behalf, etc
   c. Both also allow a state variable to be passed through the sequence…it can be set to a unique string and used to validate that connections are coming from a valid client to the app on the server side
   d. One more problem, the third party needs to be able to verify that the incoming request really is from a valid app
   e. Oauth uses a two-step process for authentication of users
   f. We'll use Twitter as an example (https://dev.twitter.com/web/sign-in/implementing)
   g. The app first asks Twitter for a request token, passing an Authorization header that identifies the app
   h. Twitter responds with a request token
   i. Next the user hits the Twitter login, passing the ID of the app (this is often via a redirect from the app back end) and the request token
   j. Twitter logs the user in and returns a verifier key along with the original request token — check the token on the server
   k. App requests an access token from Twitter (exchanging the request token for it)
   l. Twitter responds with access token and secret, which the app should store

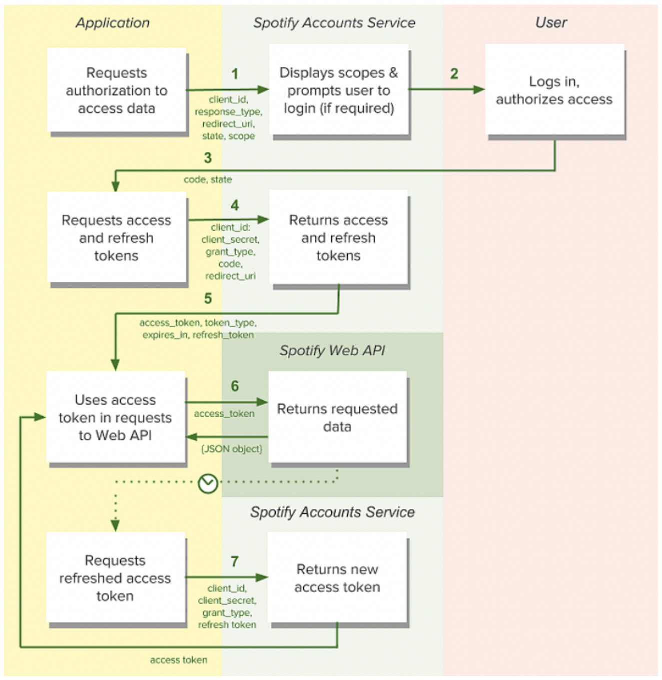
14. Application Settings

a. 

Application Settings

Your application's Consumer Key and Secret are used to authenticate requests to the Twitter Platform.

Access level	Read and write (modify app permissions)
Consumer Key (API Key)	xT0JrIMSWVc2kA4kZDdia9DE3 (manage keys and access tokens)
Callback URL	http://127.0.0.1:3000/auth/callback
Callback URL Locked	No
Sign in with Twitter	Yes
App-only authentication	https://api.twitter.com/oauth2/token
Request token URL	https://api.twitter.com/oauth/request_token
Authorize URL	https://api.twitter.com/oauth/authorize
Access token URL	https://api.twitter.com/oauth/access_token

b. 
Application
Spotify Accounts Service
User
Requests authorization to access data
1
client_id, response_type, redirect_uri, state, scope
Displays scopes & prompts user to login (if required)
2
Logs in, authorizes access
3
code, state
Requests access and refresh tokens
4
client_id, client_secret, grant_type, code, redirect_uri
Returns access and refresh tokens
5
access_token, token_type, expires_in, refresh_token
Spotify Web API
Uses access token in requests to Web API
6
access_token
{JSON object}
Returns requested data
Spotify Accounts Service
Requests refreshed access token
7
client_id, client_secret, grant_type, refresh token
Returns new access token
access token

15. Using Packages for OAuth

a. OAuth can get complicated, especially OAuth 1.0a (example: https://developer.twitter.com/en/docs/basics/authentication/guides/authorizing-arequest)
b. Every framework has packages to deal with this complexity
c. For example, Passport for Node, oauth-ruby, python-oauth2
d. These abstract OAuth and provide a simpler interface
e. Don't forget to place your OAuth app credentials and secrets in a config file that is NOT uploaded to github

16. Combine JWT and OAuth
    a. A very clean way to handle third-party authentication would be
        i. Use OAuth to authenticate users
        ii. On the callback from the third party auth provider, store user info in a local database (name, permissions, user ID, and so on)
        iii. Generate a JWT with a key that can be used to retrieve a user document from the local database
        iv. Pass the JWT on the cookie, marked httpOnly, back to the user's browser
        v. Now on each request the JWT is sent along, too, and can be verified on the back end to allow access to protected endpoints
        vi. Alternatively, send the JWT to the user's browser and (assuming you have JavaScript running there) persist it to localstorage to hold the login between browser sessions
17. A Few tips
    a. All of the client's interaction should be with the app server; that way you shouldn't run into CORS / CSRF problems
    b. If you are storing JWTs or other auth info on the cookie, be sure to delete the cookie when the user logs out of the app
    c. Most OAuth providers set a short expiration period on the access token….you can use a 'refresh' token to request a new one if the old one has expired (many devs always call for a refresh)
    d. If your app is using 'general' data from a third party like Twitter, you might not need user-specific authorization and can instead use app-specific authorization; the flow is pretty much the same with different endpoints or header flags
        i. (That doesn't mean you can't still use OAuth for user authentication)