Tema 1, parte 4
Autenticación en APIs REST

Contenidos

1. Autenticación con sesiones
2. HTTP Basic
3. Tokens
4. OAuth

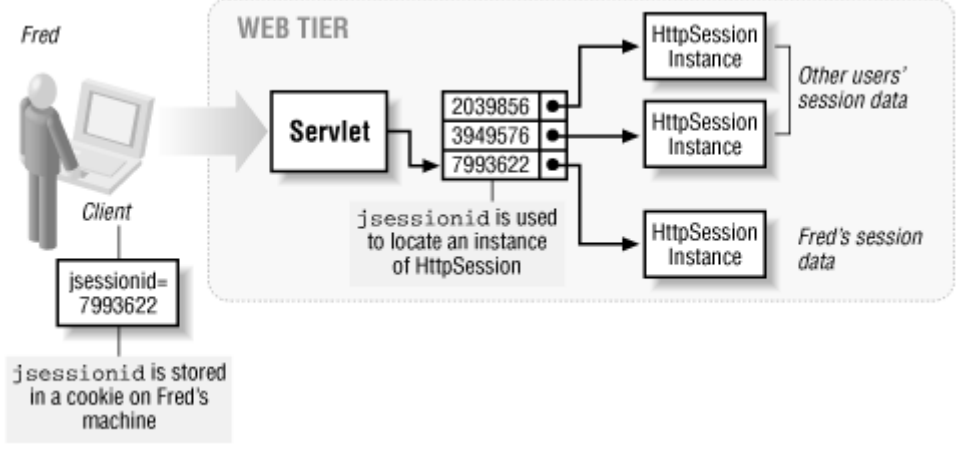
1. Autenticación en apps web "clásicas" vs. REST

Sesiones

- Aunque HTTP es originalmente un protocolo sin estado, la mayoría de aplicaciones web clásicas guardan estado en el servidor
- **Sesión**: conjunto de datos que "se recuerdan" mientras vamos navegando entre páginas (usuario autentificado, carro de la compra, ...)

Cómo funcionan las sesiones

- Sin entrar en detalles, cada cliente envía automáticamente al servidor en cada petición una *cookie* con un valor único (que generó el servidor en la primera petición)
- Mediante ella podemos identificar al cliente, y guardar datos de él en el servidor (credenciales, carro de la compra, preferencias, ,...)

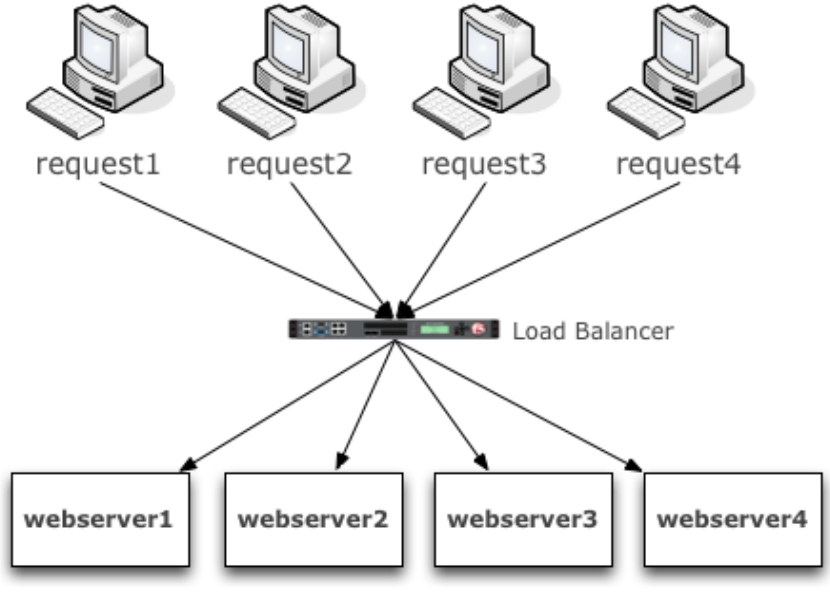


A favor de las sesiones

- Todos los lenguajes del lado del servidor ofrecen soporte
- En el lado del cliente no hay que hacer nada especial, al usar *cookies*, que están soportadas en todos los navegadores de forma automática

En contra de las sesiones

Escalabilidad: es mucho más fácil escalar una app con servidores *stateless*. Al cliente no le importa qué instancia sirva las peticiones, podemos redirigirlas, arrancar nuevos servidores, parar los que ya hay, etc.



En contra de las sesiones

"filosofía" **REST**: el API no debe recordar el estado, el estado actual se *transfiere* con cada petición

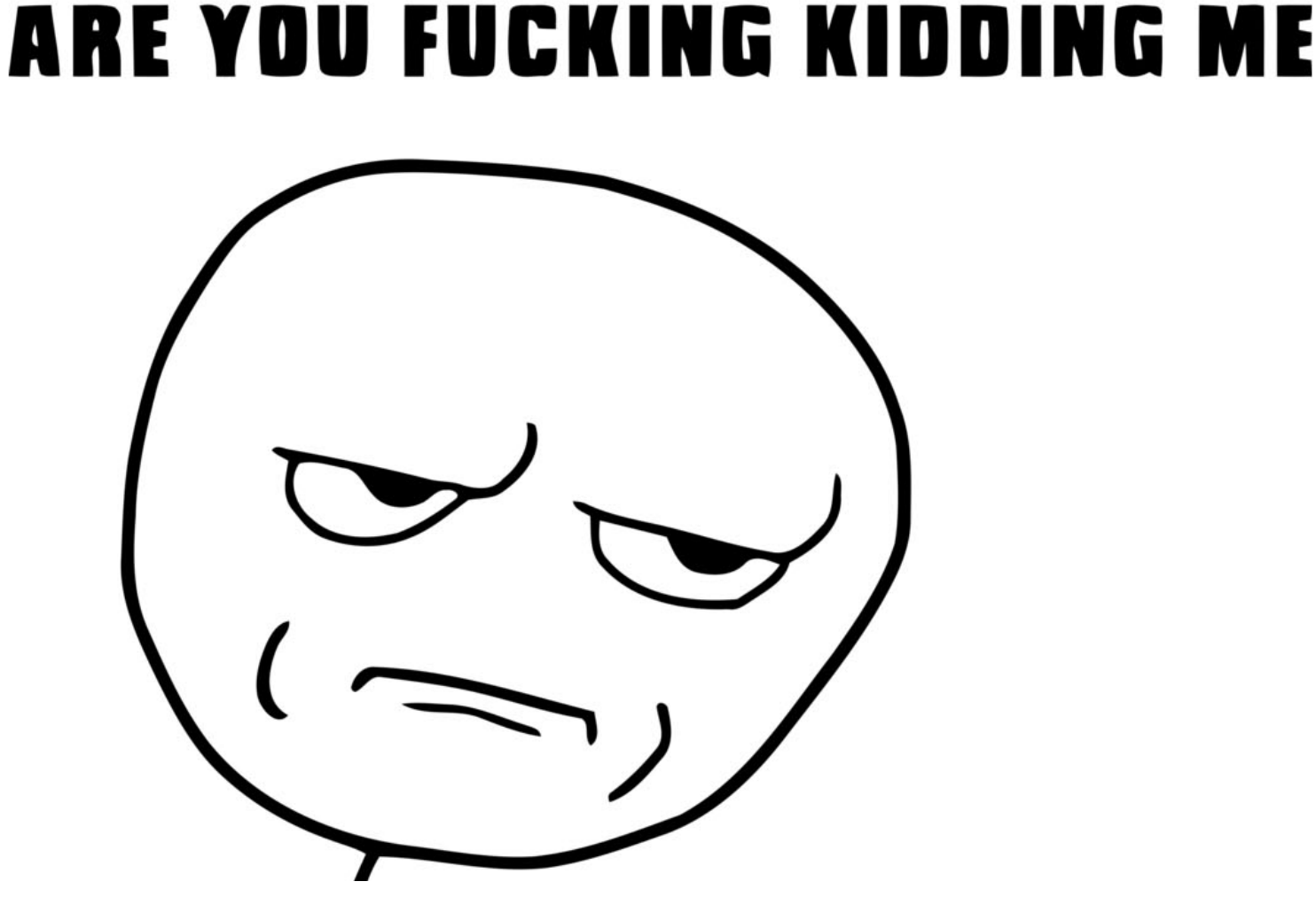
2. Autenticación con HTTP Basic

HTTP Basic

- Mecanismo **estándar** de autenticación en HTTP
- Como HTTP **no tiene estado** hay que enviar las credenciales **en cada petición**.
- Se envía **login** y **password** en Base64 (==¡sin cifrar!), separados por ":" dentro de la cabecera `Authorization`

```
Authorization: Basic cGVwaXRvOjEyMzQ1Ng==
```

¿De verdad cada vez que se realice una operación protegida el usuario tiene que introducir login y password?



HTTP Basic en una *app*. Qué ve el usuario final

1. El usuario introduce *login* y *password* en un formulario, y se hace una llamada al API simplemente para **comprobar que son correctos** (el API debería ofrecer esta operación)
    - Si son OK, se **almacenan en el navegador** (típicamente con un API muy sencillo de usar llamado *Local Storage*)
    - Si son incorrectos se muestra error
2. Como las credenciales están almacenadas en el navegador, con Javascript podemos adjuntarlas en cada petición al API (veremos ejemplos en el tema 2)

Intento de acceso sin credenciales

Según la especificación HTTP cuando se intenta acceder a un recurso protegido sin **`Authorization`**, el servidor debería responder con un *status* 401 y una cabecera **`WWW-Authenticate`**

```
401 UNAUTHORIZED HTTP/1.1
...
WWW-Authenticate: Basic realm="nombre del realm"
```

Cuando el navegador recibe un `401` + cabecera `WWW-Authenticate` hace que "salte" el típico cuadro de diálogo de login

Si no queremos que aparezca, habrá que "saltarse" el estándar (*status* distinto de 401 u obviar la cabecera `WWW-Authenticate`)

A favor de HTTP Basic

- **Estándar** HTTP
- Sencillo de implementar
- Es **stateless**, no obliga a mantener estado en el servidor

En contra de HTTP Basic

- Login y password se transmiten sin cifrar. Por tanto hay que usar **HTTPS**. Una mejora es **HTTP Digest**, que hace un *hash* MD5 de los datos.
- Si la seguridad se ve comprometida hay que cambiar el *password*, lo que es tedioso para el usuario.

3. Autenticación con *tokens*

Tokens

1. Cuando se hace *login* el servidor nos devuelve un **token** (valor idealmente único e imposible de falsear)
2. Para cualquier operación restringida debemos **enviar el token en la petición**

Un "token" en el mundo real

JSON Web Token (JWT)

- Estándar IETF. Hay implementación en multitud de lenguajes.
- Es una cadena en formato JSON formada por 3 partes:
    1. **Cabecera**: indica el tipo de token y el algoritmo de firma. Se codifica en Base64. Ejemplo: `{"typ":"JWT", "alg":"HS256"}` (indica que esto es un "JWT" y se firmará con HMAC SHA-256)
    2. **Payload**: lo que queremos almacenar en el token en formato JSON (p.ej. `{"login":"adi"}`) y codificado en Base64URL
    3. **Firma**: se aplica un algoritmo de *hash* sobre la cabecera, el payload y una clave secreta que solo conoce el servidor y se pasa a Base64URL
    4. Las tres partes se concatenan con '.'

```
eyJ0eXAiOiJKV1QiLCJhbGciOiJIUzI1NiJ9.eyJob2xhIjoibXVuZG8ifQ.pJPDprjxsouVfaaXau-Fyspj6r
```

Comprobar si un JWT es auténtico

- El servidor toma la cabecera y el *payload* (recordar que no están cifrados, solo en Base64) y la clave secreta, y vuelve a aplicar el *hash*. Si no coincide con la firma, el token no es válido.
- En teoría no se puede generar un token si no se conoce la clave secreta, y esta no se puede averiguar a partir de un token auténtico (el *hash* no es invertible)
- Recordar que, **todo se transmite "en claro"**: Base64 es una codificación, no un cifrado. Por tanto normalmente habrá que usar HTTPS si no se quiere que el *payload* sea legible

Fecha de expiración

- En el *payload* se suele incluir una fecha de expiración del *token*. En el estándar se especifica el uso de **`exp`** (con el nº segundos desde el 1/1/1970). Si el *token* ya ha expirado el servidor debería devolver el *status* 401
- De paso solucionamos el problema de que el mismo *payload* siempre genera el mismo JWT si no cambiamos el *secret*

Ejemplo en Node.js

```
var jwt = require('jwt-simple');
var moment = require('moment');  //para trabajar cómodamente con fechas

var payload = {
    login: 'pepito',
    exp: moment().add(7, 'days').valueOf()
}

var secret='123456';

//crear el JWT a partir de payload + secret
var token = jwt.encode(payload, secret);
console.log(token);

//validar el JWT. "decode" comprueba que sea válido y nos devuelve el payload
var decoded = jwt.decode(token, secret);
if (decoded) {
  console.log("¡¡Token válido!!. Payload: " + decoded);
}
//en realidad el payload se puede sacar decodificando Base64
//lo importante del método "decode" es que chequea la firma
//Con 'split' partimos el token por los ".", devuelve un array de Strings
```

[https://runkit.com/ottocol/ejemplo-jwt](https://runkit.com/ottocol/ejemplo-jwt/2.0.0)

"Flujo" de uso de JWT

1. El **cliente presenta** las credenciales (normalmente **login+password**) al servidor y a cambio obtiene un token JWT. En el estándar no se especifica en qué parte de la petición/respuesta colocar la información. Puede ser p.ej. en el cuerpo, en formato JSON
2. En **cada operación restringida hay que enviar el JWT**. Se debería hacer en la cabecera `Authorization`. Se pone la palabra clave `Bearer` seguida del JWT.

```
Authorization: Bearer eyJ0eXAiOiJKV1QiLCJhbGciOiJIUzI1NiJ9...
```

JWT en una app para el usuario final

1. El usuario introduce su *login* y *password* en un formulario, y se hace una llamada al servidor para **obtener un *token***. Es decir, el API debe implementar esta operación.
    - Si es OK, el JWT se almacena en el navegador (típicamente en el *local storage*)
2. **Con cada llamada** "protegida" al API **adjuntamos el *token***
    - Si el *token* no es correcto o ha expirado, el servidor devolverá `401`. En caso de expiración, si teníamos almacenado *login/password* la *app* puede pedir un nuevo JWT de manera transparente al usuario

¡Cuidado!: almacenar información sensible como *tokens* JWT en el "local storage" podría acarrear problemas de seguridad ya que otro código JS en la misma página tiene acceso a él (susceptible a vulnerabilidad XSS o *Cross-Site Scripting*)

- https://stormpath.com/blog/where-to-store-your-jwts-cookies-vs-html5-web-storage
- https://stackoverflow.com/questions/44133536/is-it-safe-to-store-a-jwt-in-localstorage-with-reactjs

La solución más común es almacenar el JWT en *cookies* de tipo `httpOnly`, que no se pueden leer desde JS pero se envían automáticamente al servidor con cada petición como todas las *cookies*

A favor de los *tokens*

Con respecto a HTTP Basic

- En caso de información comprometida, es mucho menos engorroso invalidar un *token* que hacer que el usuario cambie su *password*

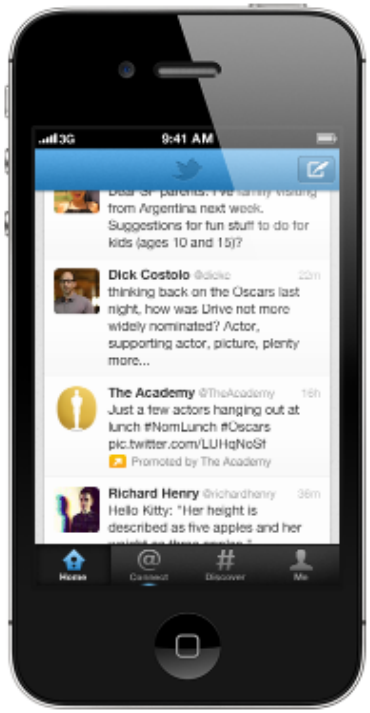
Con respecto a las sesiones HTTP (recordad, usan *cookies*)

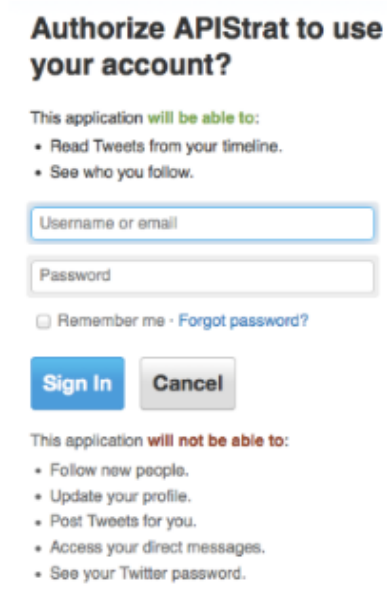
- Se pueden usar también en aplicaciones nativas (escritorio, móviles)
- El dominio del servicio de autenticación puede ser distinto al del API

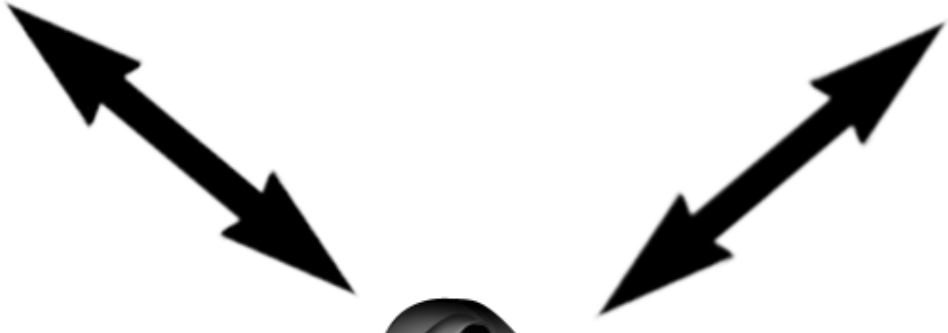
4. OAuth

¿Qué es OAuth?

- Protocolo de autenticación diseñado originalmente para cuando nuestra aplicación quiere acceder al API de un tercero pero el usuario no está dispuesto a confiarnos su *password*
- Solución: el usuario se autentifica directamente con el API del tercero y este nos cede un *token*, válido durante un tiempo. Si hay problemas es mucho más sencillo anular el *token* que cambiar el password


9:41 AM
from Argentina next week. Suggestions for fun stuff to do for kids (ages 10 and 15)?
Dick Costolo @dickc 22m
thinking back on the Oscars last night, how was Drive not more widely nominated? Actor, supporting actor, picture, plenty more...
The Academy @TheAcademy 16h
Just a few actors hanging out at lunch #NomLunch #Oscars pic.twitter.com/LUHqNoSf
Promoted by The Academy
Richard Henry @richardhenry 38m
Hello Kitty: "Her height is described as five apples and her
Home
Connect
Discover
Me


Authorize APIStrat to use your account?
This application will be able to:
Read Tweets from your timeline.
See who you follow.
Username or email
Password
Remember me · Forgot password?
Sign In
Cancel
This application will not be able to:
Follow new people.
Update your profile.
Post Tweets for you.
Access your direct messages.
See your Twitter password.



Aunque OAuth fue diseñado sobre todo para el caso de uso anterior, la versión actual contempla más casos de uso (**grant types** en el argot), como por ejemplo una aplicación hecha por los "propietarios" del API y en la que sí podemos confiar nuestro *password* (el *grant type* llamado *Resource Owner Password Credentials*)

Como resultado, OAuth se ha convertido en el **estándar de autenticación "de facto"** en APIs REST de acceso público

Más sobre OAuth

- La versión actual es la 2, una simplificación de la original, que tenía fama de ser muy complicada
- Tutoriales interesantes para más información:
  - *Mitchell Anicas*, An introduction to OAuth 2
  - *Aaron Parecki*, OAuth 2 simplified

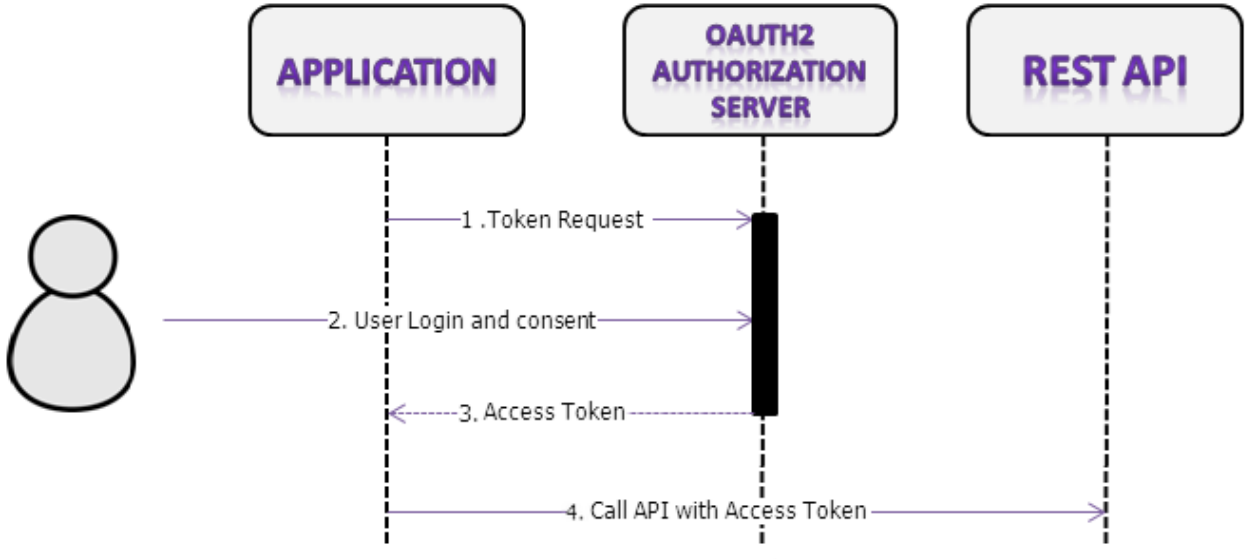
¿OAuth vs. JWT?

En realidad no son comparables. OAuth es un *protocolo* que indica cómo se deben comunicar cliente/servidor en el proceso de autenticación, JWT es un *mecanismo* de autenticación.

De hecho, en OAuth podemos usar *tokens* JWT ya que OAuth no especifica cómo generar los *token*.

Ejemplo con Facebook

- Ejemplo del denominado *implicit grant* según OAuth 2
- Como condición previa debemos tener una app dada de alta en FB como desarrolladores. Dicha app tiene un id único.



Documentación de FB

1. Se hace una petición a **`https://www.facebook.com/dialog/oauth?client_id={id_de_la_app}&redirect_uri={redirect_uri}&response_type=token`**
2. Podemos añadir a la URL anterior el param. **`scope`** con permisos solicitados por la *app*, por ejemplo **`scope=email,publish_actions`**
3. FB muestra una página de login para nuestra app
4. Se hace una redirección a la **`redirect_uri`**. Dentro de la URL de la redirección aparecerá un **`access_token`** con un *token* para acceder al API.
5. Para cualquier petición al API se debe usar un parámetro HTTP **`access_token`** con el *token* obtenido. Por ejemplo

```
//ver información sobre mi
GET https://graph.facebook.com/me?access_token={token_obtenido}
//enviar un mensaje a mi muro
POST graph.facebook.com/{user-id}/feed?message={message}&access_token={token_obtenido}
```

¿Preguntas...?