Approndimenti su HTTP: cookie, autenticazione, CORS e Caching

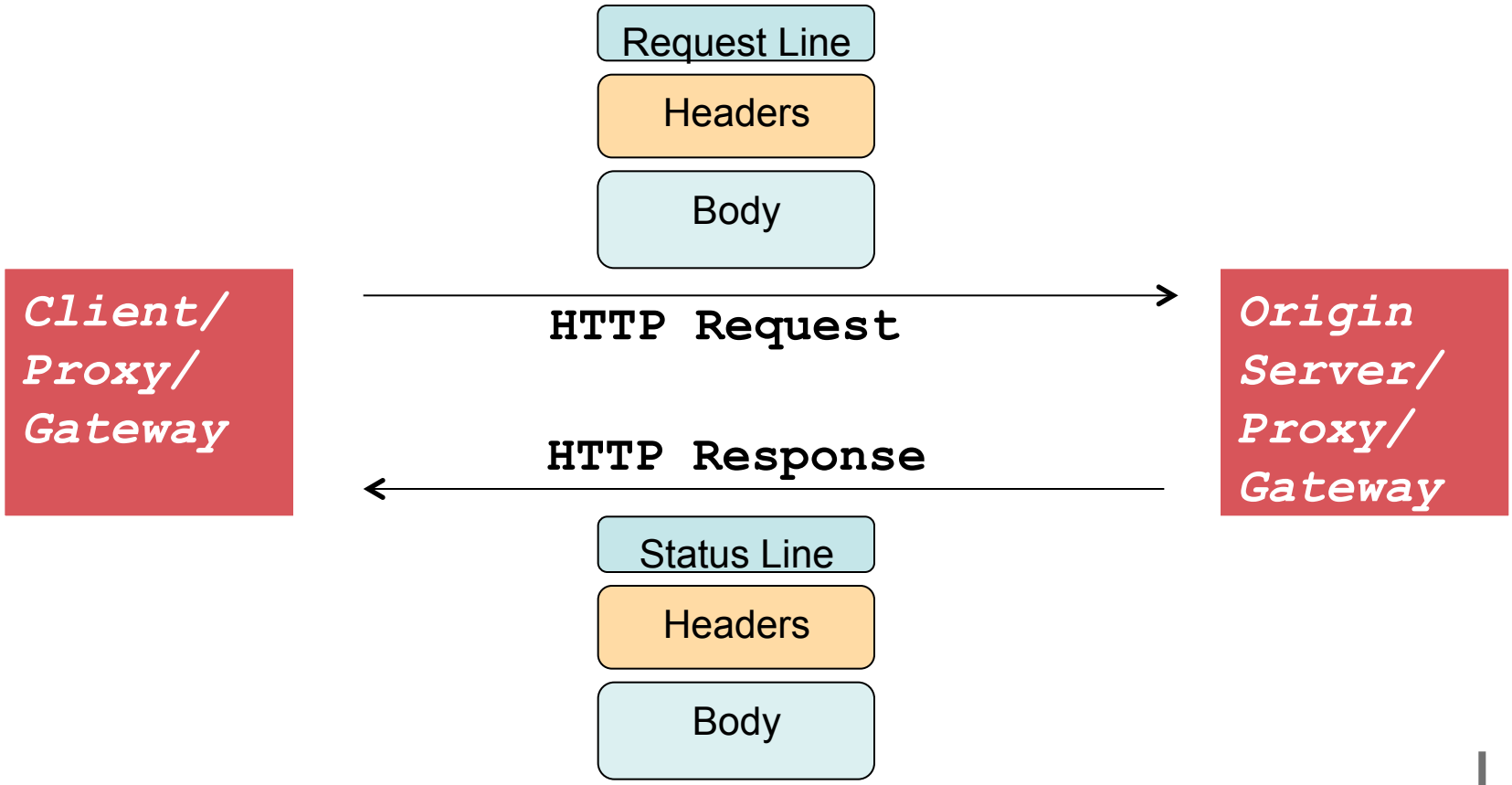
Angelo Di Iorio

Università di Bologna

Sommario

- Oggi parleremo di:
  - Cookies
  - Autenticazione
    - HTTP Authentication
    - JWT
  - CORS
  - Caching

Recap



I cookies

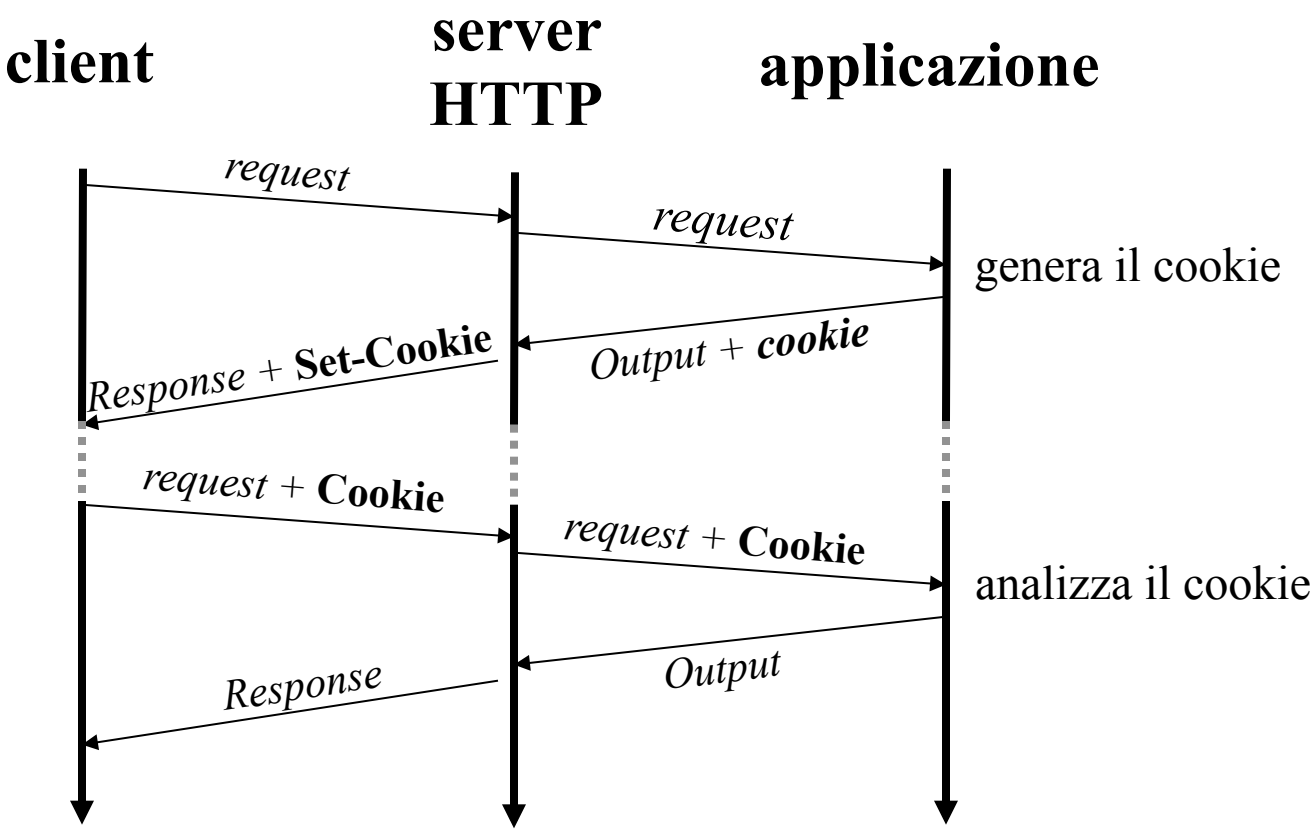
• HTTP è **stateless**: non esiste nessuna struttura ulteriore alla connessione e il server non è tenuto a mantenere informazioni su connessioni precedenti

• Un cookie (non in HTTP, è un'estensione di Netscape, proposta nell'RFC 2109 e poi ancora RFC 2965) è una breve informazione scambiata tra il server ed il client

• Il client mantiene alcune informazioni sulle connessioni precedenti, e le manda al server di pertinenza ogni volta che richiede una risorsa

• Il termine *cookie* (anche *magic cookie*) indica un blocco di dati opaco (i.e.: non interpretabile) lasciato in consegna ad un richiedente per poter ristabilire in seguito il suo diritto alla risorsa richiesta (come il tagliando di una lavanderia)

Architettura dei cookies (1)



Architettura dei cookie (2)

• Alla prima richiesta di uno user-agent, il server fornisce la risposta ed un header aggiuntivo, il **cookie**, con **dati arbitrari**, e con la specifica di usarlo per ogni successiva richiesta.

• Il server associa a questi dati ad informazioni sulla transazione.

• Ogni volta che lo user-agent accederà a questo sito, rifornirà i dati opachi del cookie che permettono al server di ri-identificare il richiedente

• I cookies dunque usano due header, uno per la risposta, ed uno per le richieste successive:

- **Set-Cookie**: header della risposta, spedito dal server, il client può memorizzarlo e rispedirlo alla prossima richiesta.
- **Cookie**: header della richiesta. Il client decide se spedirlo sulla base del nome del documento, dell'indirizzo IP del server, e dell'età del cookie.

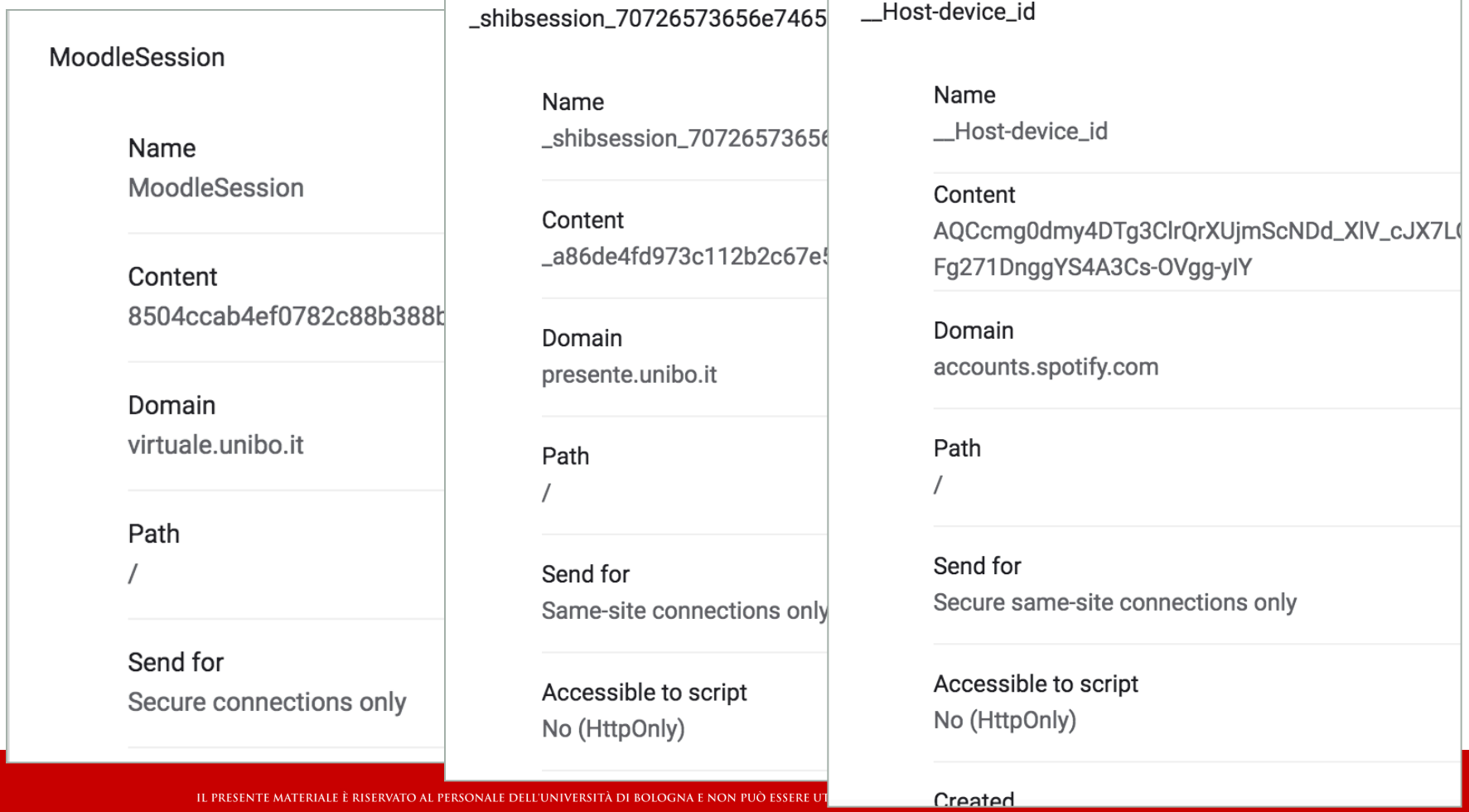
Headers e struttura dei cookie

- I cookies contengono dati arbitrari in formato testuale:

```
set-cookie:AWSALB=oJW4vtjxMz8WLY3jSrCUNekkG1aLZQHmblSvExmv
R6agqb+f+fC4RQWCb+gLVijpRakI8RrnfrXqiDmQ9KwqA8LiVMdhkBRUCCtO
gwx5JBxKLmIBQ7gnbIFIzo+; Expires=Wed, 01 May 2000 11:07:46
GMT; Path=/
```

- Oltre a nome e valore, includono informazioni che il client usa quando ri-spedisce il cookie:
  - **Domain**: il dominio per cui il cookie è valido
  - **Path**: l'URI per cui il cookie è valido
  - **Max-Age/Expire**: La durata in secondi del cookie
  - **Secure**: la richiesta che il client contatti il server usando soltanto un meccanismo sicuro (es. HTTPS) per spedirlo
  - **Version**: La versione della specifica a cui il cookie aderisce.

Esempi



Uso dei cookie

- Esitono vari tipi di cookie usati per scopi diversi, tra cui:
  - **Cookie persistenti**: hanno una validità temporale lunga, o addirittura non scadono mai, e sono usati per mantenere informazioni (semi)permanenti ad esempio informazioni su login, preferenze degli utenti, ecc.
  - **Cookie di sessione**: hanno una durata breve e sono usati per raggruppare operazioni in sessioni di lavoro; contengono un identificativo della sessione le cui informazioni sono sul server; solitamente sono cancellati alla chiusura del browser
  - **Cookie di terze parti**: appartengono ad un dominio diverso rispetto a quello della pagina in cui sono caricati; usati ad esempio per banner pubblicitari, permettono di ricostruire la navigazione degli utenti e possono essere usati in modo improprio; le impostazioni dei browser permettono infatti di disabilitarli

Autenticazione

- Uno dei campi di maggiore applicazione dei cookie è l'autenticazione
- Molto spesso infatti l'accesso a risorse (o servizi) Web è ristretto ad uno o più utenti che devono quindi essere riconosciuti
  - **Autenticazione**: processo di verifica dell'identità di un utente
  - **Autorizzazione**: processo di verifica dell'effettiva possibilità di un utente di eseguire un'operazione e/o accedere a una risorsa
- Si vuole fare in modo che, una volta autenticato l'utente non debba ripetere questa operazione per le successive richieste (fino alla scadenza della validità)

Autenticazione: sessioni o token

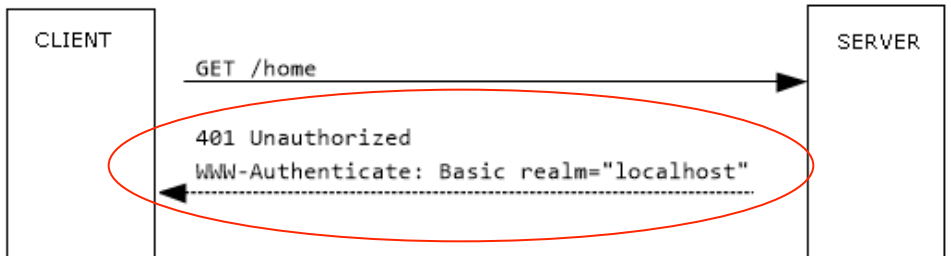
- Per aggregare e collegare più richieste, previa autenticazione, si usano due approcci principali:
  - **Session-based**: il server memorizza un ID di sessione e informazioni sull'utente verificate ad ogni richiesta
  - **Token-based**: il client memorizza un token, ricevuto dal server, che spedisce ad ogni richiesta ed è usato per la verifica
- Entrambi si possono realizzare sfruttando specifici header HTTP e i cookie

Header *WWW-Authenticate*

- Quando l'utente prova ad eseguire un'azione che richiede autenticazione, il server risponde con uno status code **`401`** (*Unauthorized*) e aggiunge un header **`WWW-Authenticate`** in cui specifica i criteri da usare per l'autenticazione

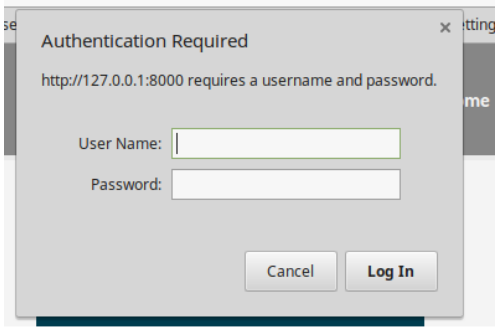
- Il meccanismo è generico e permette di usare criteri diversi per spedire i dati di autenticazione, anche personalizzati (ad esempio ne esiste uno specifico per AWS, Amazon Web Services)

- Gli schemi più comuni
  - ***Basic*** e ***Digest***
  - ***Bearer***, usato anche per autenticazione *token-based*



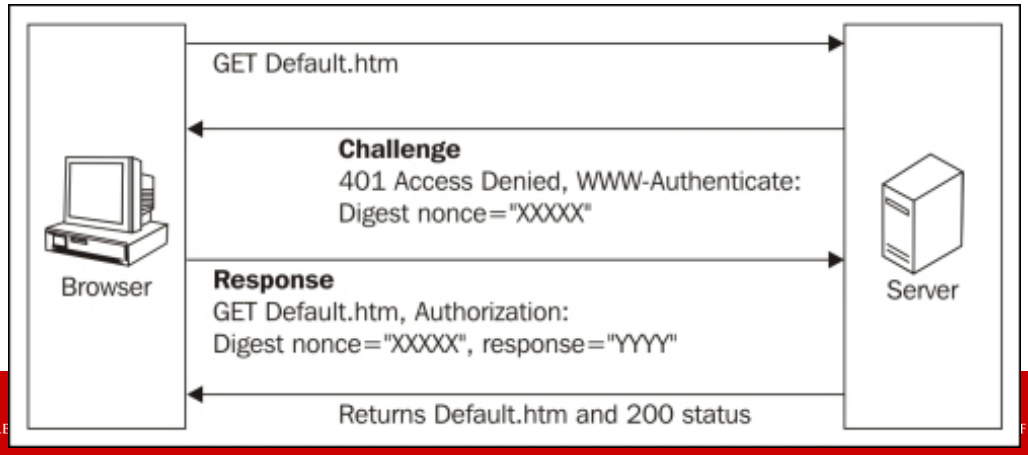
Schema Basic

- Il meccanismo basilare, ormai in disuso, prevede la spedizione delle informazioni di autorizzazione in chiaro ad ogni risposta
  - Il server indica il contesto di sicurezza (detto "Realm") nella prima risposta tramite l'header **`WWW-Authenticate`**
  - Il client chiede all'utente le informazioni di autorizzazione, crea una nuova richiesta GET e fornisce le informazioni di autorizzazione codificate in *Base64* nell'header **`Authorization`**.
  - Il client continua a mandare lo stesso header per le successive richieste allo stesso *realm*
- Molte limitazioni:
  - La password passa viaggia in chiaro
  - Non è prevista un'operazione di chiusura della sessione di lavoro
  - Il form di autenticazione non è personalizzato e integrato nell'applicazione ma nel client



Schema Digest

- Per evitare di spedire la password in chiaro, è stato introdotto uno schema *Digest*
- Il client spedisce nell'header Authorization una *fingerprint* della password, ovvero la password crittografata con il metodo MD5 (RFC 1321).
- Per evitare *replay attack* il server spedisce al client anche una stringa causale (nonce) che viene crittografata dal client insieme alla password
- Il server decrifra i dati ricevuti dal client e se corretti lo autentica
- L'operazione è ripetuta ad ogni richiesta



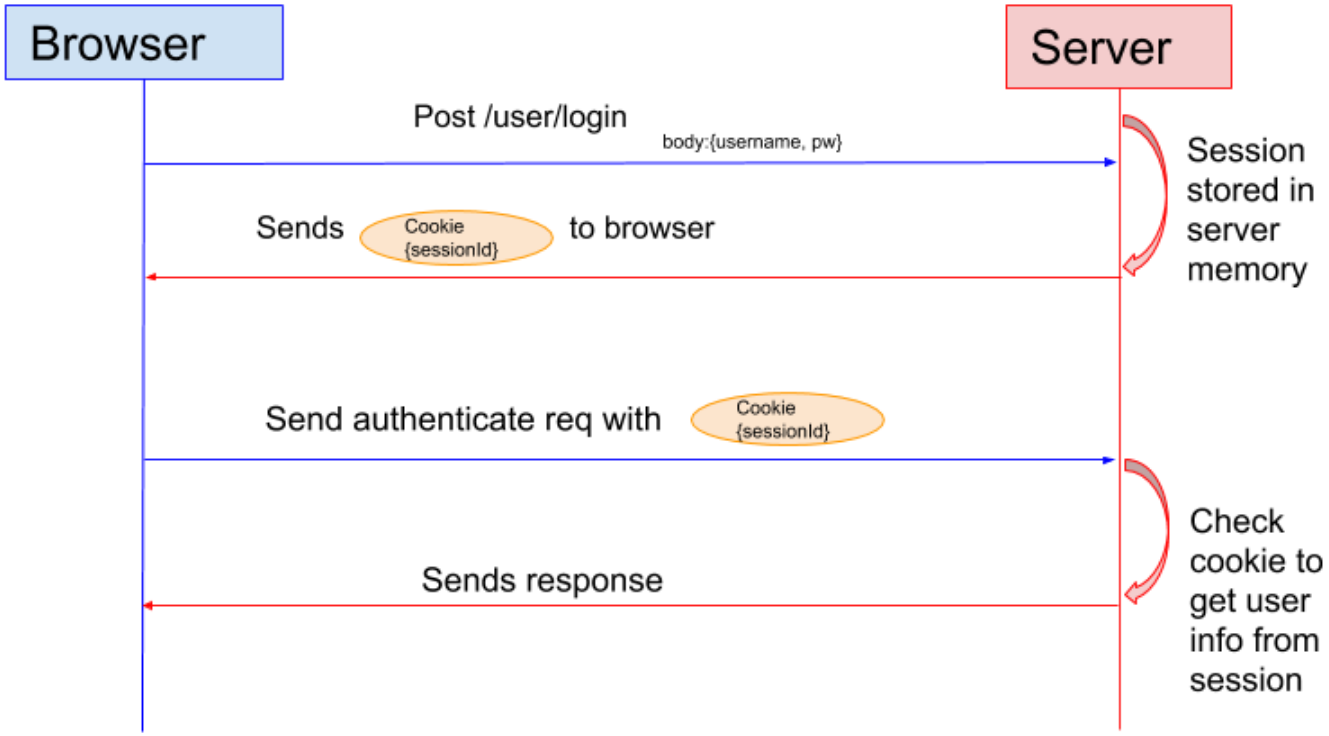
Schema Bearer

- Nello schema Bearer (RFC 6750) il client non spedisce la password, in chiaro o cifrata, ma un token ("bearer token") che gli permette di accedere ad una risorsa
- Il token è stato precedentemente generato dal server, controllando l'identità del client, ma il solo possesso è sufficiente per essere autenticati
- E' importante quindi fare in modo che questi token non siano falsificati
- Il meccanismo è generico e viene usato per l'autenticazione **token-based,** in contrapposizione a quella **session-based**

Session-based authentication (1)

- Una *sessione* è un insieme di azioni "collegate" ed eseguite in un dato intervallo di tempo
- Dopo aver verificato l'identità dell'utente, il server:
  - genera una **sessione** a cui **associa un ID** e le informazioni relative all'utente; **memorizza le informazioni** sulla sessione
  - spedisce al client l'ID della sessione appena avviata
- Nelle successive richieste il client spedisce ID della sessione e altre informazioni per autenticarsi
- Il server verifica queste informazioni, inclusa la loro validità temporale (una sessione può scadere e richiede quindi di ripetere il processo di autenticazione) e restituisce la risorsa all'utente
- Esistono diversi modi per spedire le informazioni di sessione, tra cui i cookie

Session-based authentication (2)



Cookie di sessione

- I *cookie di sessione* contengono un ID univoco di sessione che il server ha memorizzato e riconosce
- Il server si occupa anche di gestire la data di *expiry* (scadenza) del cookie e quindi chiusura della sessione
- Tutti i linguaggi di programmazione server-side hanno una gestione automatica dei cookie di sessione e permettono allo sviluppatore di leggere facilmente le informazioni collegate

Esempio in PHP

```php
<?php
   session_start();

   if( isset( $_SESSION['counter'] ) ) {
      $_SESSION['counter'] += 1;
   }else {
      $_SESSION['counter'] = 1;
   }

   $msg = "You have visited this page ".  $_SESSION['counter'];
   $msg .= "in this session.";
?>
```

Esempio in Express.js

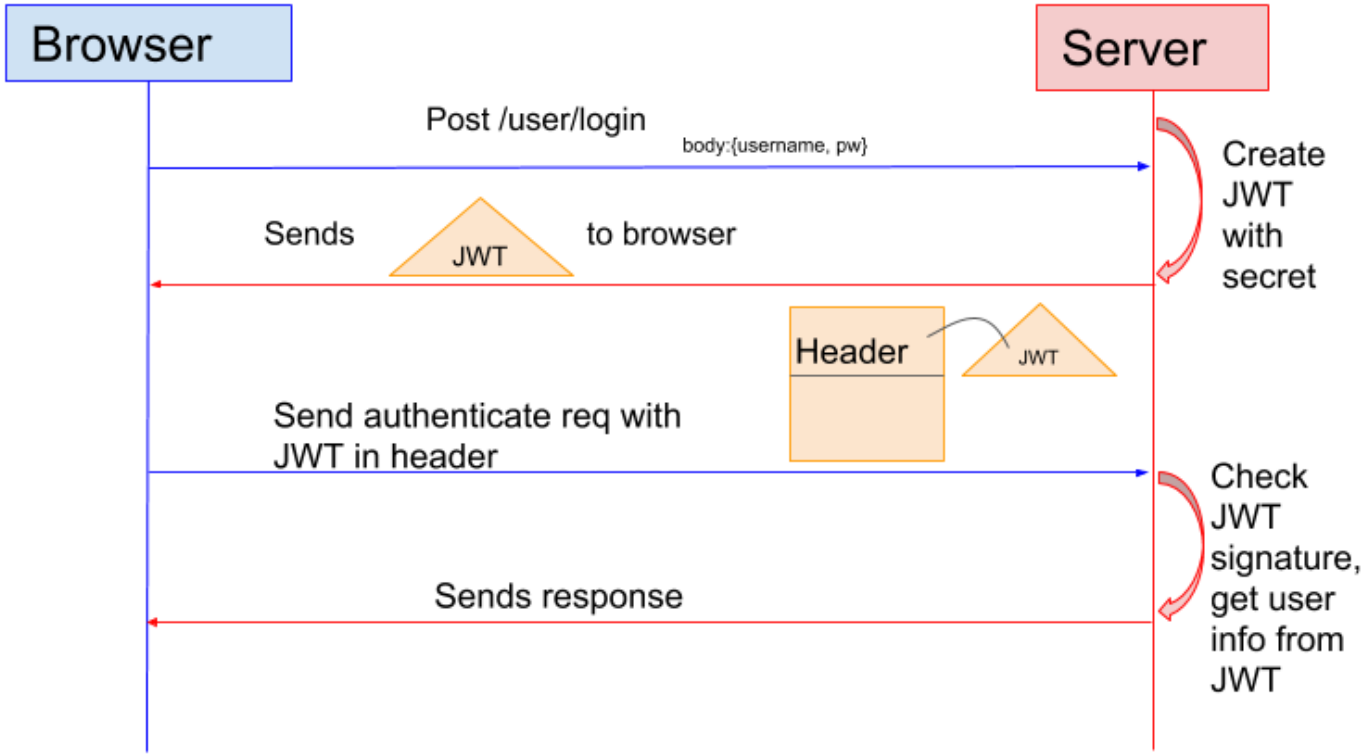
```
// Use the session middleware
app.use(session({ secret: '<secret>', cookie: { maxAge: 60000 }}))

// Access the session as req.session
app.get('/', function(req, res, next) {
  if (req.session.views) {
    req.session.views++
    res.setHeader('Content-Type', 'text/html')
    res.write('<p>views: ' + req.session.views + '</p>')
    res.write('<p>expires in: ' + (req.session.cookie.maxAge / 1000) + 's</
    res.end()
  } else {
    req.session.views = 1
    res.end('welcome to the session demo. refresh!')
  }
})
```

Token-based Authentication

- Un approccio alternativo, e largamente usato oggi perché più scalabile, prevede l'uso di token e non richiede di memorizzare sul server informazioni relative alle sessioni
- Dopo aver autenticato l'utente, il server crea un token con un segreto, lo firma con la propria chiave e lo spedisce al client
- Il client memorizza il token e lo aggiunge negli header (tipicamente Authorization, con lo schema Bearer) di tutte le successive richieste
- Il server verifica la propria firma sul token e le informazioni sull'utente e, se valide, restituisce la risorsa
- Lo schema è generico, il token può essere di vari tipi così come il meccanismo usato per firmarlo

Token-based Authentication (con JWT)



Sessioni o token?

	Sessioni server-side	Token
Informazioni sulla "sessione"	Server	Client
Riduzione carico e uso di memoria server-side		X
Riduzione dati da memorizzare		X
Maggiore controllo su revoca ed expiry	X	
Meno rischi di contraffazione	X	

JWT – JSON Web Token

- JWT è uno standard (RFC 7519) che definisce un formato JSON per lo scambio di token di autenticazione, in generale di informazioni (detti ***claims***) tra servizi Web
- Il meccanismo è generico e permette di:
  - Personalizzare i claim
  - Usare algoritmi diversi per firmare i messaggi
- Si basa a sua volta su altri standard per firmare (JSON Web Signature) e cifrare (JSON Web Encyption) messaggi in formato JSON
- Sintassi *compatta* e *URL-safe*

Un passo indietro: Base 64

Input	M								a								n							
Codice ASCII	77								97								110							
mappa bit	0	1	0	0	1	1	0	1	0	1	1	0	0	0	0	1	0	1	1	0	1	1	1	0
Indice 6-bit	19								22				5				46							
Output Base64	T								W				F				u							
Codice ASCII	84								87				70				117							

- Viene identificato un sottoinsieme di 64 caratteri di US-ASCII sicuri, che hanno la stessa codifica in tutte le versioni di ISO 646.
- Ogni flusso di dati viene suddiviso in blocchi di 24 bit (3 byte). A loro volta questi 24 bit sono suddivisi in 4 blocchi di 6 bit ciascuno e codificati secondo una tabella prefissata in uno dei 64 caratteri
- La decodifica di Base64 è algoritmica, banale, non usa chiavi né calcoli di particolari complessità. NON È una tecnica crittografica!

Struttura token JWT (1)

- Un token JWT è composto da tre parti, separate da un punto e ottenute codificando i dati di input in base64 o firmandoli: **header**, **payload** e **signature**

Header e Payload

Header: specifica il tipo di token e l'algoritmo di cifratura utilizzato

Payload: informazioni di interscambio organizzate in affermazioni (**claims**) che possono essere di tre tipi:

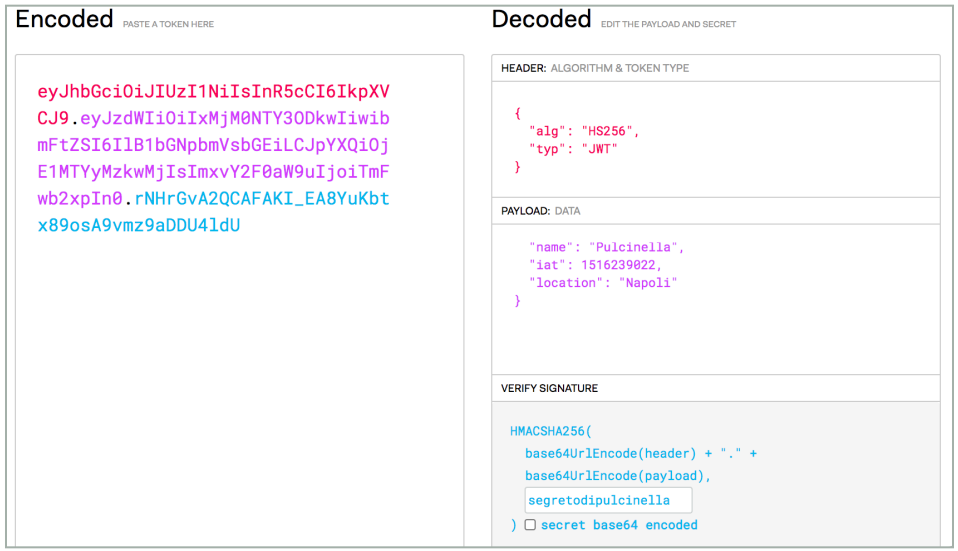
- **Registered**: claim predefiniti utili per descrivere il token
  - Entità che ha generato il token (**iss**, **issuer**)
  - Timestamp della generazione del token (**iat**, **issue at**)
  - Validità del token indicata in secondi (**exp**, **expiration time**)
  - Data dopo la quale il token inizia ad essere valido (**nbf**, **not before**)
- **Public**: arbitrari ma dichiarati nello IANA JSON Web Token Registry per evitare conflitti
- **Private**: arbitrari e personalizzabili, utili per scambiare dati tra applicazioni che si accordano sui dati da usare

Signature

- Il token (header e payload, in base64) può essere firmato con una chiave segreta lato server in modo da poterlo verificare nelle successive richieste e, se corrotto, non considerarlo valido
- Cifratura simmetrica e asimmetrica (nell'esempio è stato usato lo schema HMAC con algoritmo SHA256)
- Si possono usare diversi algoritmi, da dichiarare nell'header
- **Il contenuto del token non è cifrato ma solo codificato in base64!**
- **Si può decodificare facilmente, non bisogna quindi inserire dati sensibili**

Debugger e librerie JWT

- Validatore on-line per JWT: https://jwt.io/#debugger-io



- Esistono diverse librerie per generare e validare token JWT

Express.js e autenticazione

- In Express.js l'autenticazione è realizzata con appositi middleware
- Come negli altri casi, devono essere installati, inclusi (con `require`) e aggiunti alla propria applicazione (con `use`)
- Uno dei più usati è Passport.js:
  - **http://passportjs.org/**
  - Flessibile e modulare
  - Supporta diverse strategie di autenticazione tra cui HTTP basic e digest, e JWT
- Per aggiungere il supporto a JWT invece si usa il pacchetto **express-jwt**:
  - **https://www.npmjs.com/package/express-jwt**

CORS

Cross-site vulnerability

- Supponiamo che in una pagina appartenente ad un dominio protetto (es. in una sessione autenticata) si riesca ad intrufolare un pezzo di codice malizioso, che raccoglie e trasmette informazioni ad un repository accessibile ai malintenzionati.
- Si parla di vulnerabilità tra domini o ***cross-domain***
- Una politica (molto conservatrice) dei browser è quella di rifiutare qualunque connessione Javascript ad un dominio diverso da quello della pagina ospitante (*dominio diverso = schema o dominio o porta diversa)*. Solo gli script che originano nello stesso dominio della pagina HTML verranno eseguiti.
- Una delle possibili soluzioni, chiamata CORS, si basa sull'uso del metodo OPTION per indicare i domini ammessi

CORS

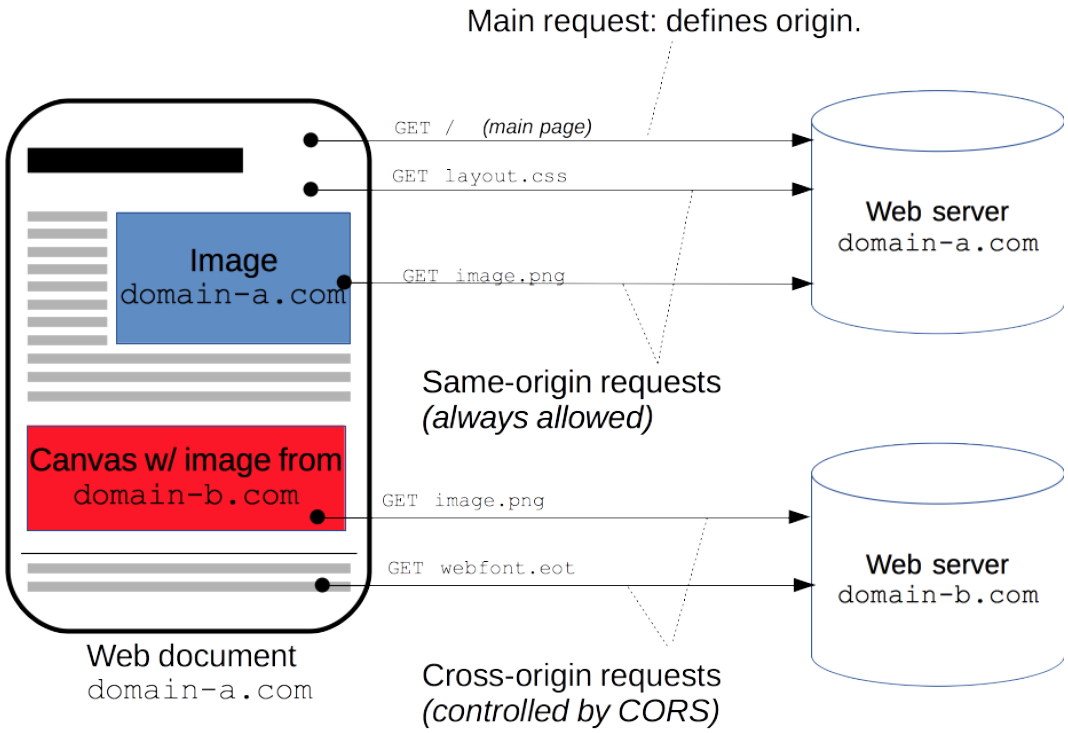


Immagine da https://developer.mozilla.org/en-US/docs/Web/HTTP/CORS

Cross-Origin Resource Sharing (CORS)

- Una tecnica tutta HTTP introdotta dal W3C (W3C Rec 29/1/2013)

- Si attiva solo per le connessioni Ajax (non si usa per connessioni HTTP normali) e prevede l'uso di due nuovi header:
  - nella richiesta: ***Origin***, per specificare il dominio su cui si trova il contenuto
  - nella risposta**:** ***Access-Control-Allow-Origin*** per indicare gli altri domini da cui è permesso caricare contenuti

- Inoltre si pone l'accento sull'uso di un **preflight** (verifica preliminare) della possibilità di eseguire un comando cross-scripting, ad esempio usando il metodo OPTIONS di HTTP.

CORS – Una sessione

- Richiesta di `www.dominio1.com` a `www.dominio2.com`
  - Metodo: GET o OPTIONS
  - `Origin: www.dominio1.com`

- Risposta di `www.dominio2.com` a `www.dominio1.com`
  - Status code: 200
  - `Access-Control-Allow-Origin:` `http://www.dominio1.com`
    - Permesse richieste cross-domain solo da http://www.dominio1.com

oppure

  - `Access-Control-Allow-Origin: *`
    - Permesse richieste cross-domain da tutti i domini

Express.js e CORS

- In Express.js si usa il middleware *cors* per aggiungere gli header **`Origin`** e **`Access-Control-Allow-Origin`** e supportare CORS
- Necessario includere il modulo e aggiungere il middleware
- Permette di abilitare le successive richieste Ajax su tutti i domìni o su domìni specifici e di specificare altre opzioni attraverso oggetti JS passati in input

```
cors = require('cors');
app.use(cors())
app.options('*', cors())
```

- Pacchetto npm: **`https://www.npmjs.com/package/cors`**

HTTP Caching

Caching

- HTTP offre sofisticati meccanismi di caching, molto utili in applicazioni RESTFUL
- Può essere client-side, server-side o su un proxy (intermedia)
- La cache server-side riduce i tempi di computazione di una risposta, ma non ha effetti sul carico di rete
- Le altre riducono il carico di rete.
- HTTP 1.1 introduce due tipi di meccanismi per controllare la validità dei dati in cache (cache control):
  - **Server-specified expiration**
  - **Heuristic expiration**

Server-specified expiration

- Il server indica una scadenza della risorsa. Può usare due meccanismi:
  - header **Expires**
  - direttiva *max-age* in **Cache-Control**
- Se la data di scadenza è già passata, la richiesta deve essere rivalidata.
- Se la richiesta accetta anche risposte scadute o se l'origin server non può essere raggiunto, la cache può rispondere con la risorsa scaduta ma con il codice 110 (Response is stale)

Cache-control

- L'header **Cache-Control** permette di controllare altri comportamenti della cache:
  - Se Cache-Control specifica la direttiva *must-revalidate*, la risposta scaduta non può mai essere rispedita. In questo caso la cache deve riprendere la risorsa dall'origin server. Se questo non risponde, la cache manderà un codice 504 (Gateway time-out)
  - Se Cache-Control specifica la direttiva *no-cache*, la richiesta deve essere fatta sempre all'origin server.

Heuristic expiration

- Poiché molte pagine non conterranno valori espliciti di scadenza, la cache stabilisce valori euristici di durata delle risorse, dopo le quali assume che sia scaduta.
- Usa informazioni contenute in altri header, ad esempio *Last-Modified time*
- L'algoritmo esatto non è fissato nelle specifiche di HTTP ma dipende dall'implementazione
- Queste assunzioni possono a volte essere ottimistiche, e risultare in risposte scorrette.
- Se non valida con sicurezza una risposta assunta fresca, allora deve fornire un codice 113 heuristic expiration alla risposta.

Validazione delle risorse in cache

- Anche dopo la scadenza, nella maggior parte dei casi, una risorsa sarà ancora non modificata, e quindi la risorsa in cache valida.
- Due modi per verificarlo:
  - usare HEAD: il client fa la richiesta, e verifica la data di ultima modifica. Ma questo richiede una richiesta in più.
  - Un modo più corretto è fare una richiesta condizionale (con appositi header): se la risorsa è stata modificata, viene fornita la nuova risorsa normalmente, altrimenti viene fornita la risposta 304 (not modified) senza body. Questo riduce il numero di richieste
    - Alcuni header: *if-Match, If-Modified-Since, If-Unmodified-Since*, ..

Conclusioni

- Oggi abbiamo approfondito alcuni aspetti collegati ad HTTP
  - Cookies
  - Autenticazione
  - CORS
  - Caching