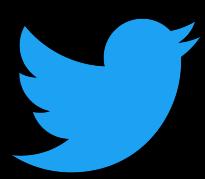


Authentication in single-page applications

Michele Stieven

Angular GDE, Founder of accademia.dev



@MicheleStieven

Menu

- Authentication
- Tipologie di attacchi nel web (XSS, CSRF)
- Authorization
 - Access Token, Refresh Token
 - JWT
- OAuth 2.0
- OpenID Connect
- Strategie custom

Authentication



Confermare l'identità di qualcuno, sulla base di credenziali

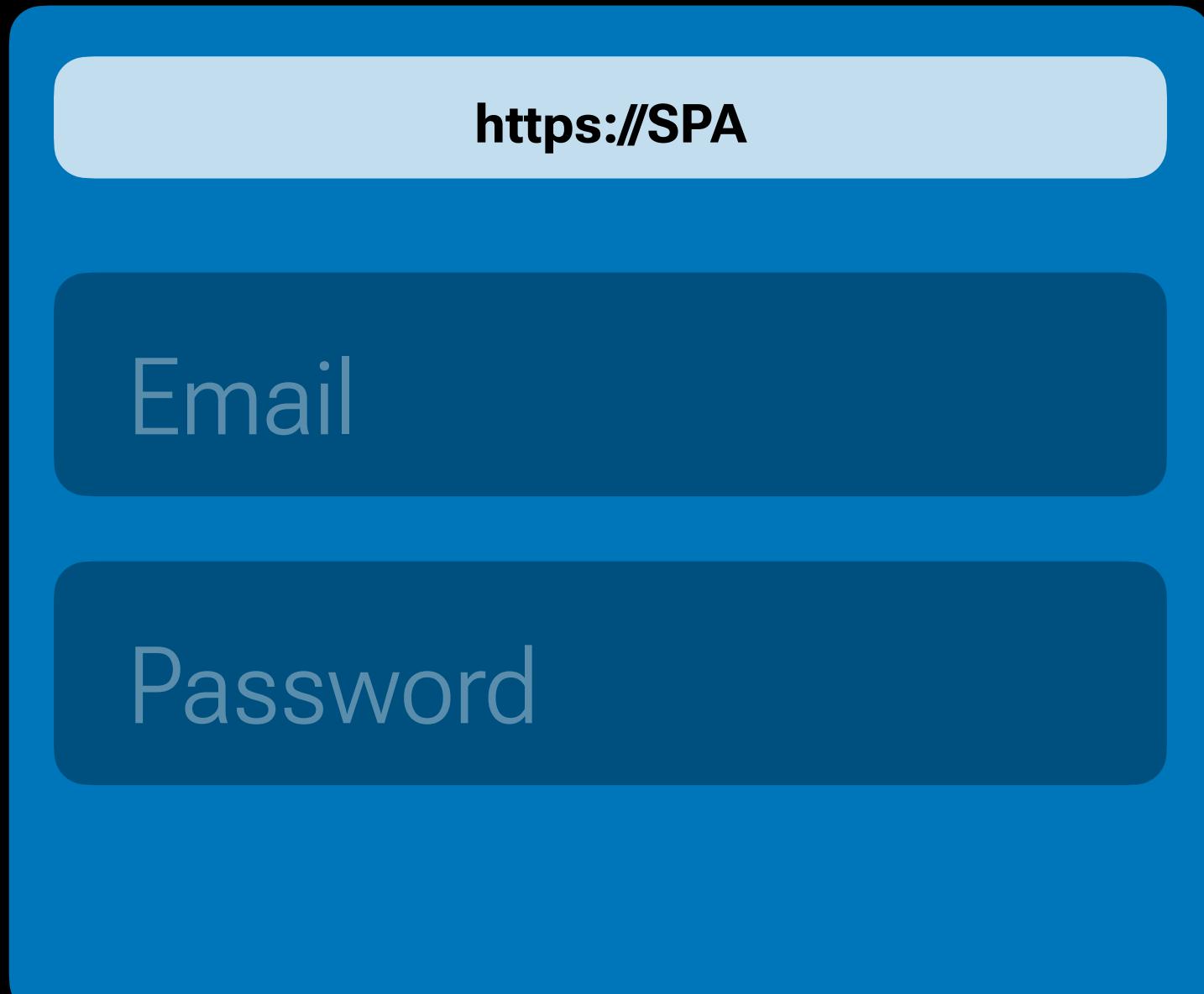
- Email e password
- Username e password
- PIN
- Impronta digitale
- Scansione della retina

Cookie Authentication

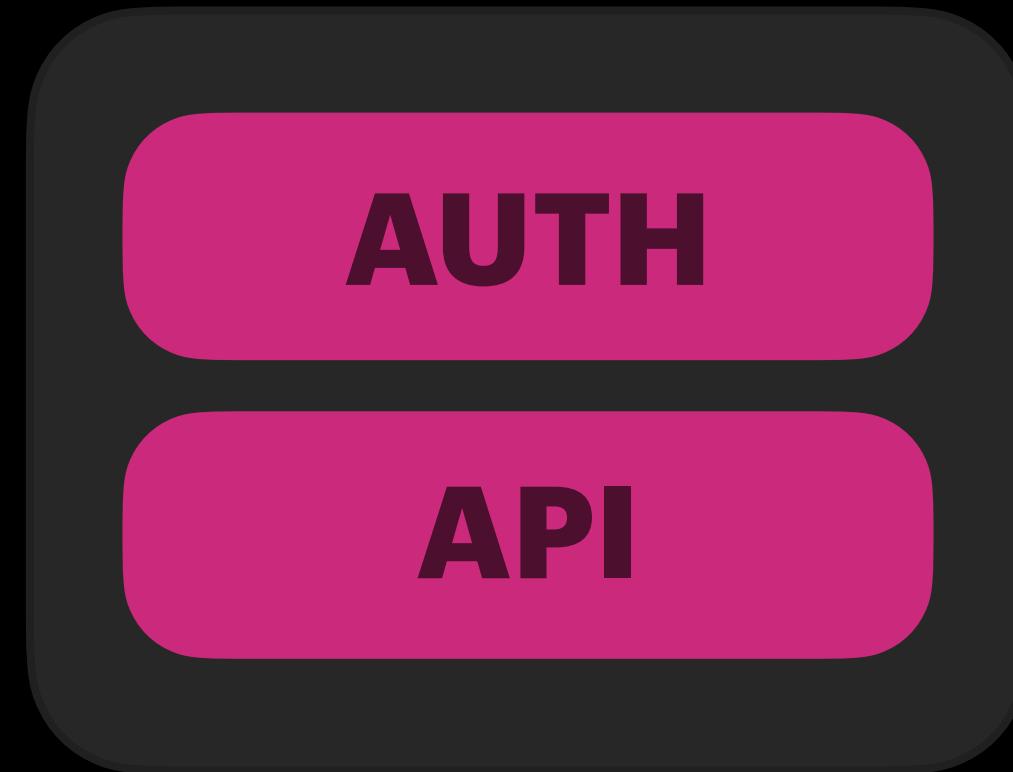


Semplice, funzionale... Lo standard sul Web

Client



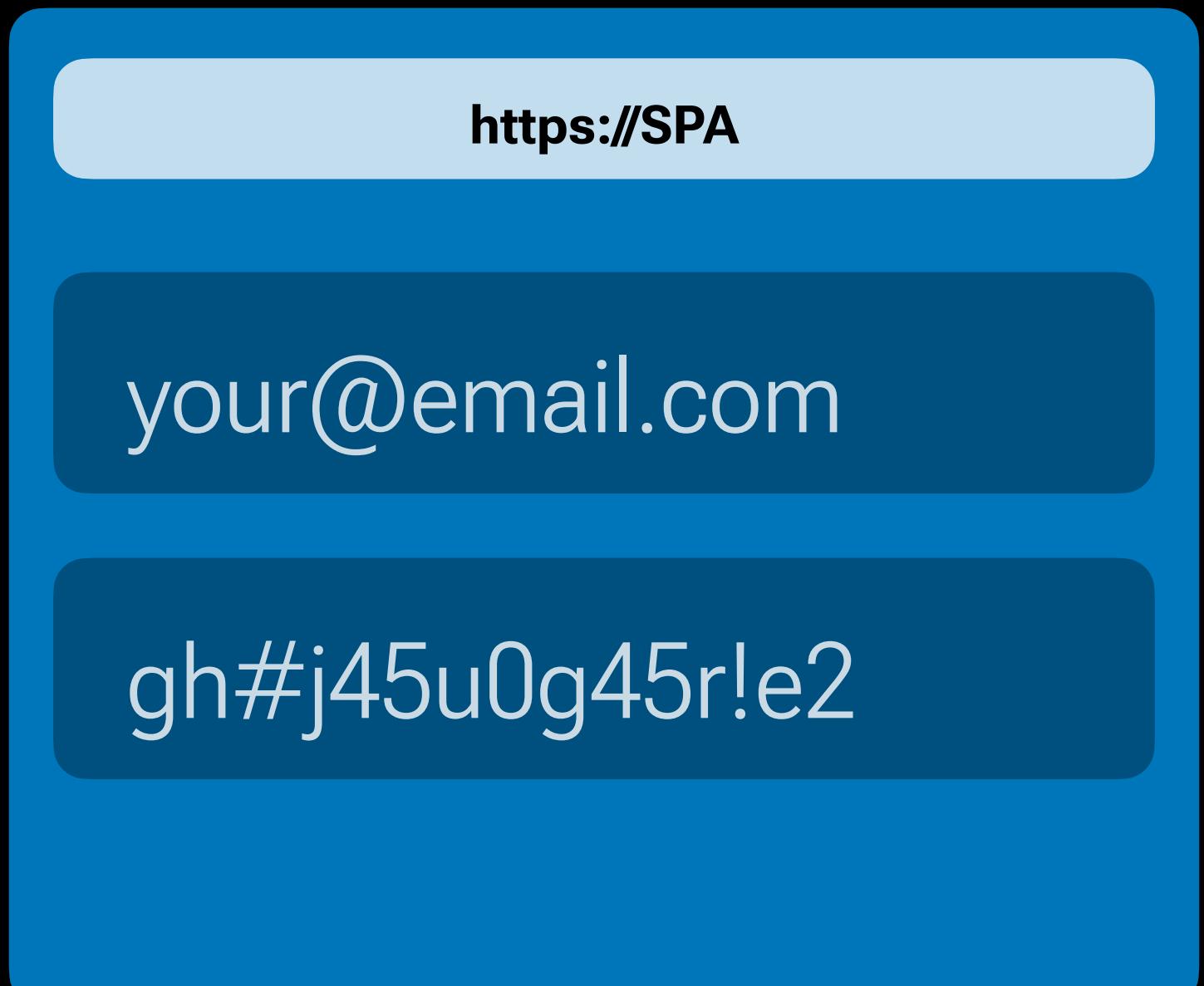
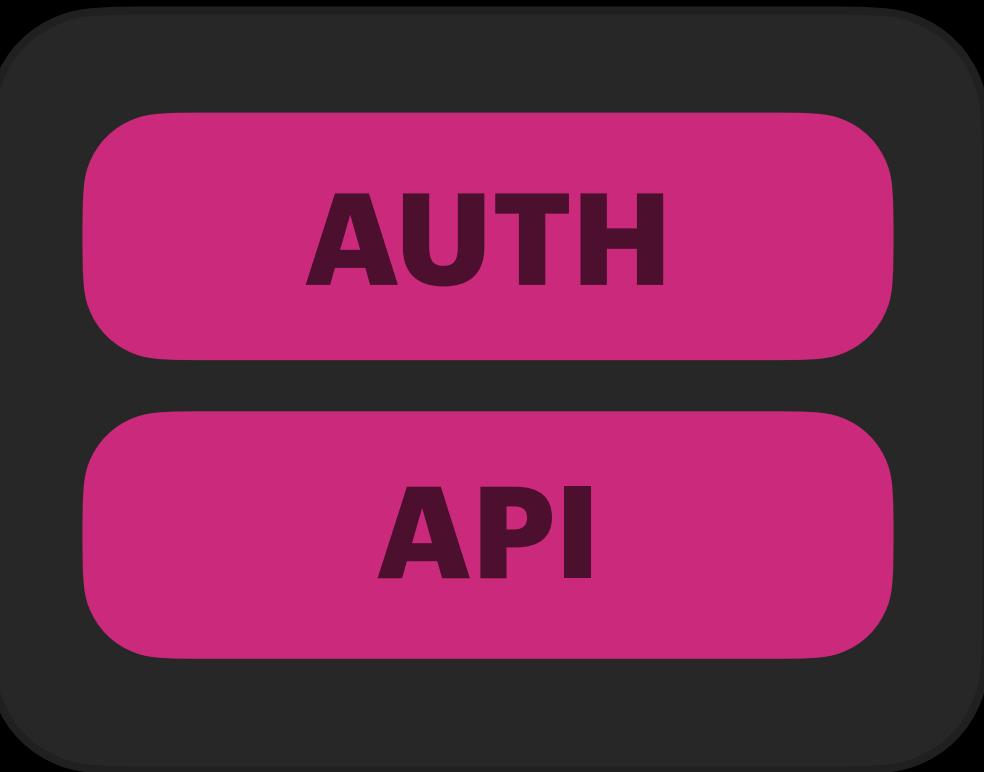
Server



Client



Server



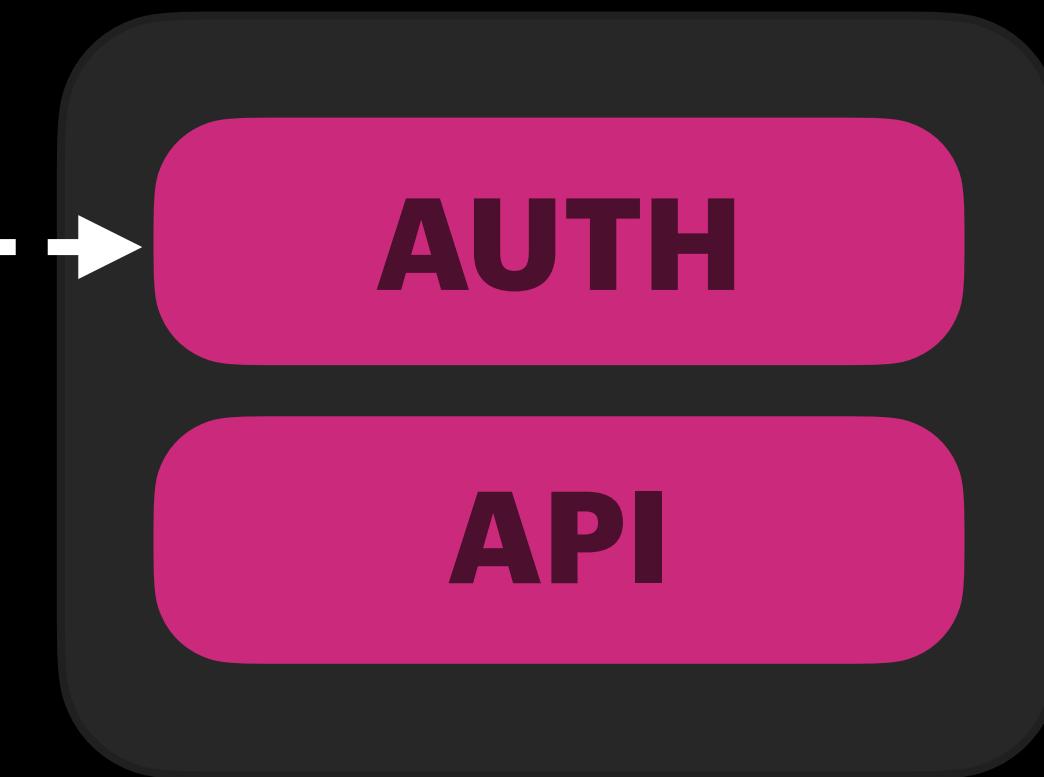
Client



Authentication

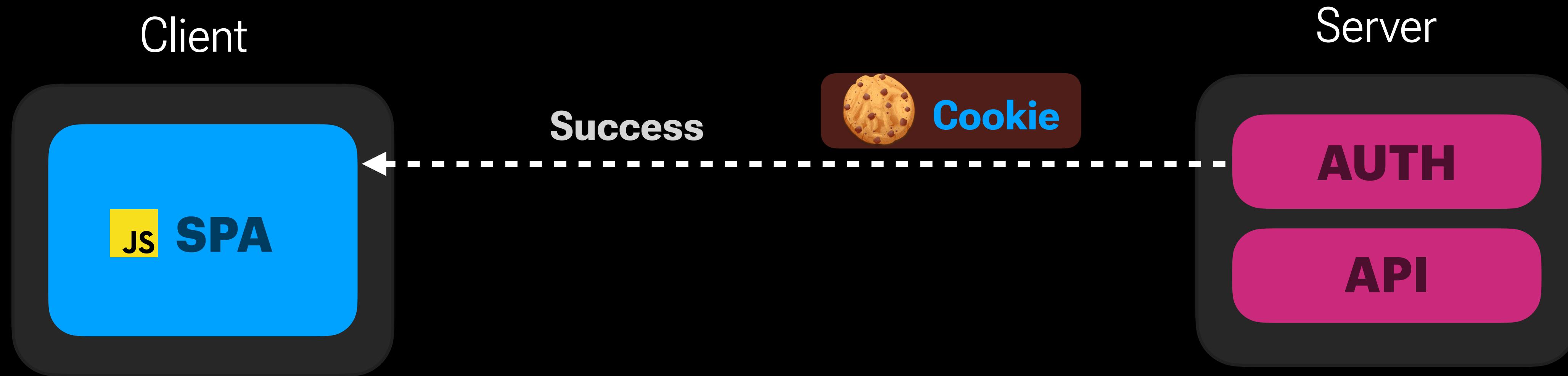
email, password

Server



AUTH

API



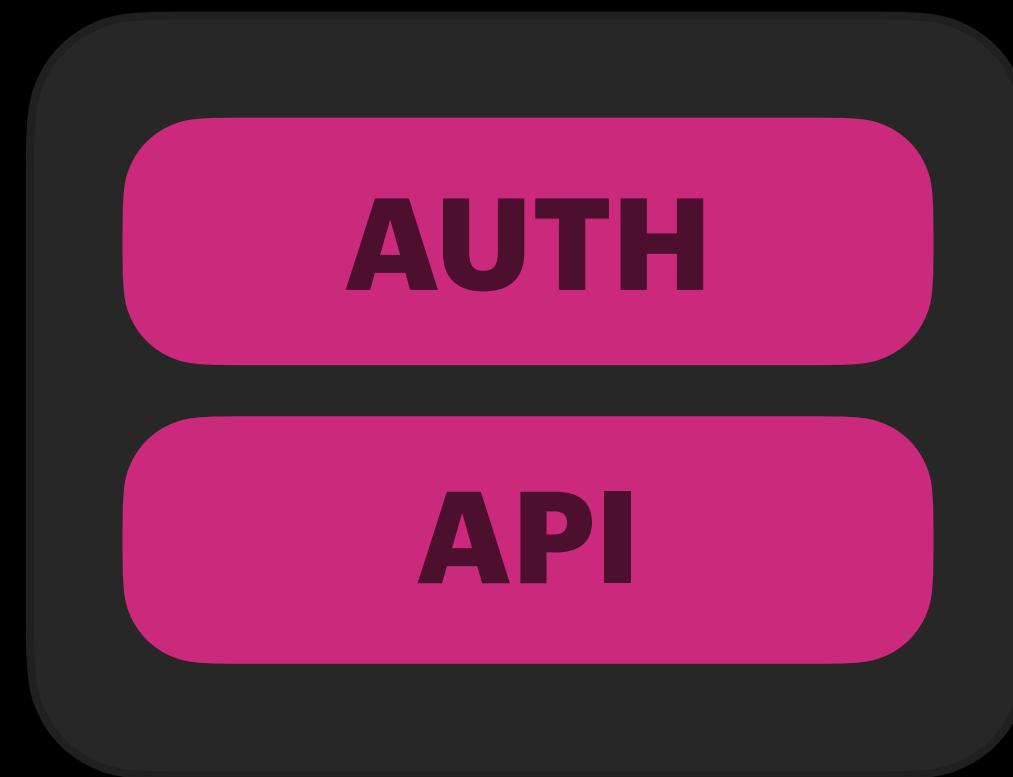
Ogni risposta di un server può settare nel tuo browser dei Cookie!

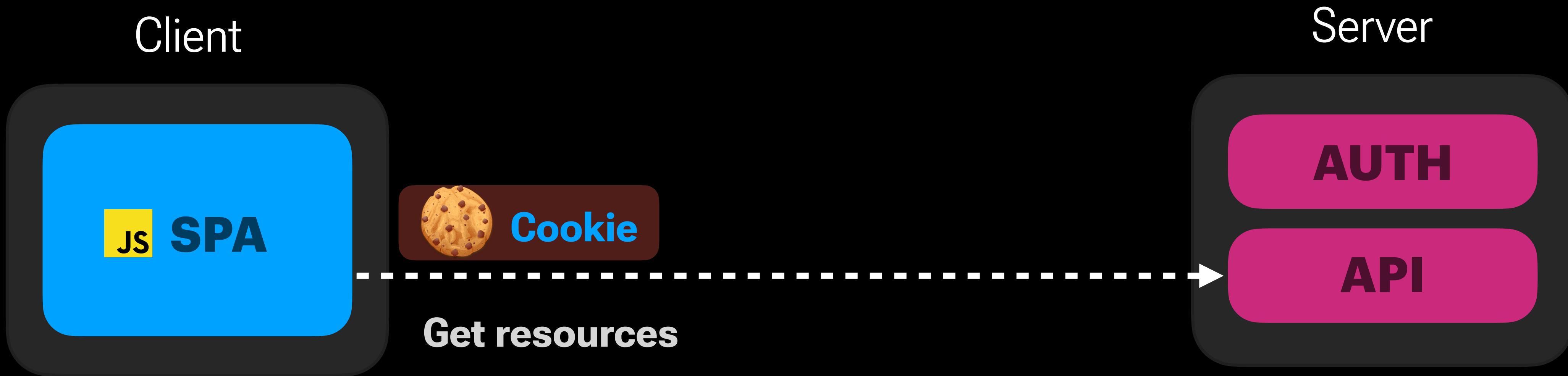
Client



Cookie

Server



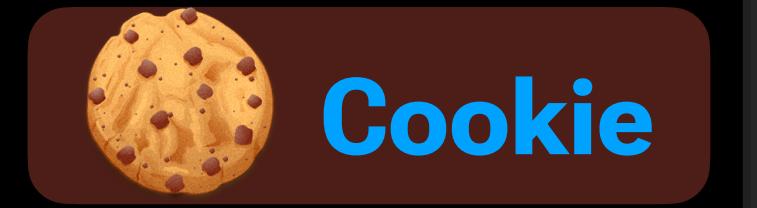




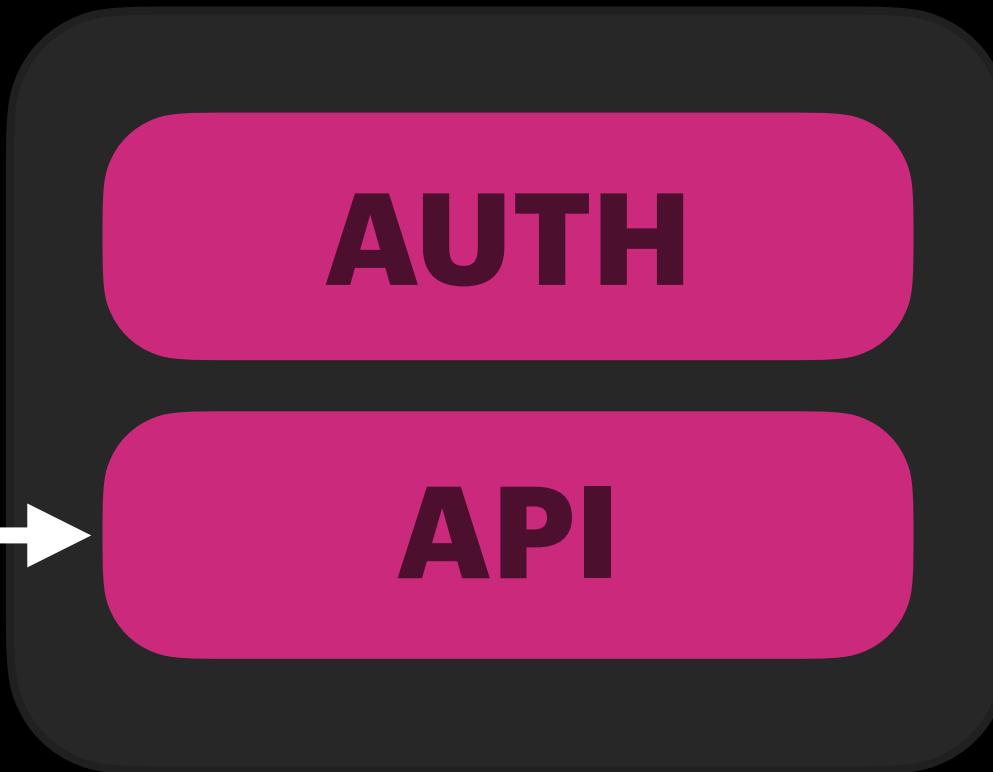
Client



/profile
/me
/userinfo



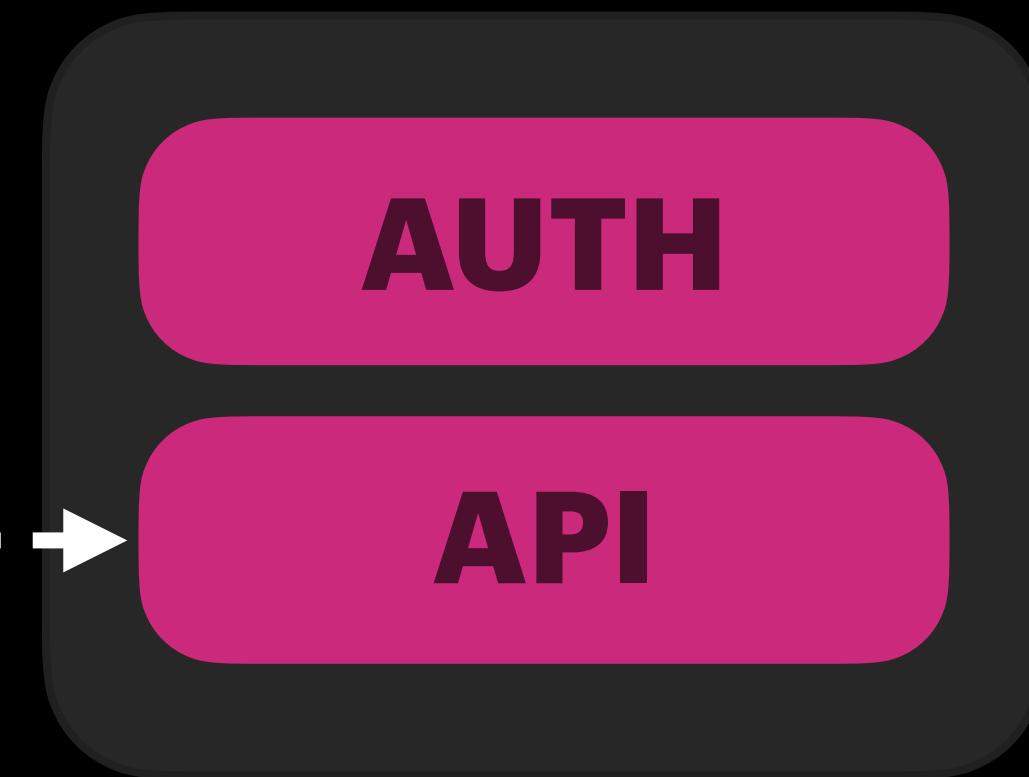
Server



Client



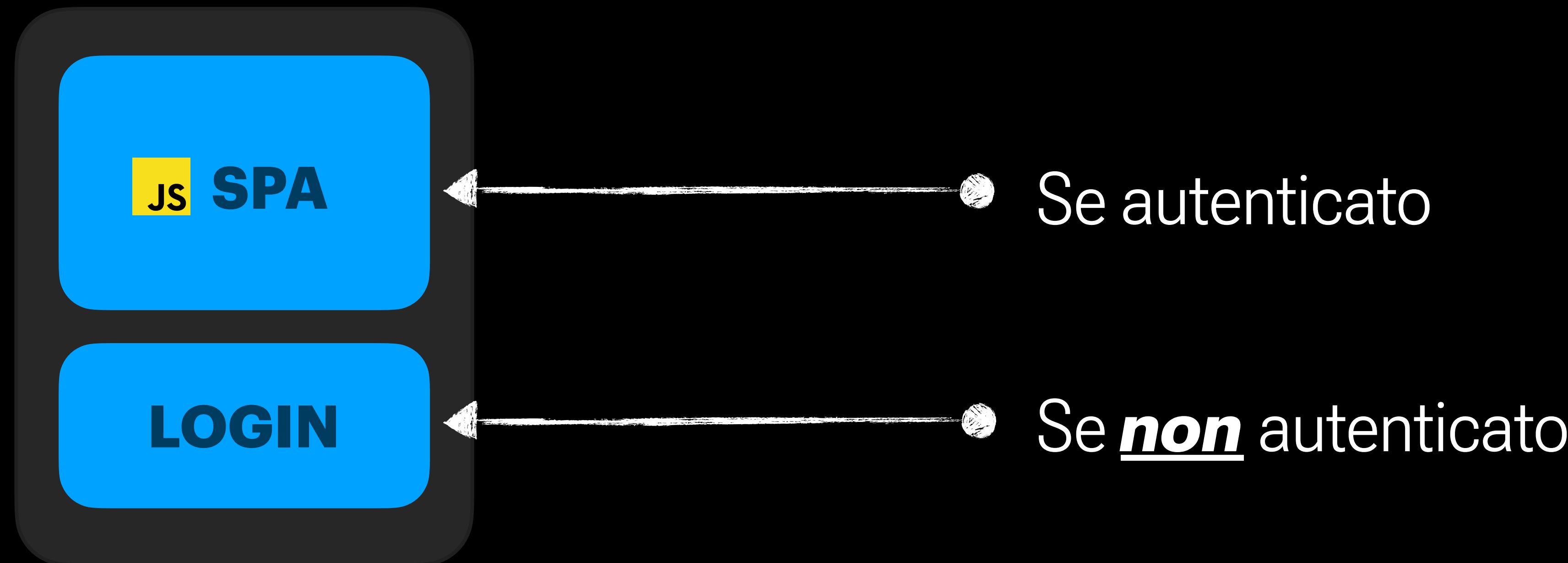
Server



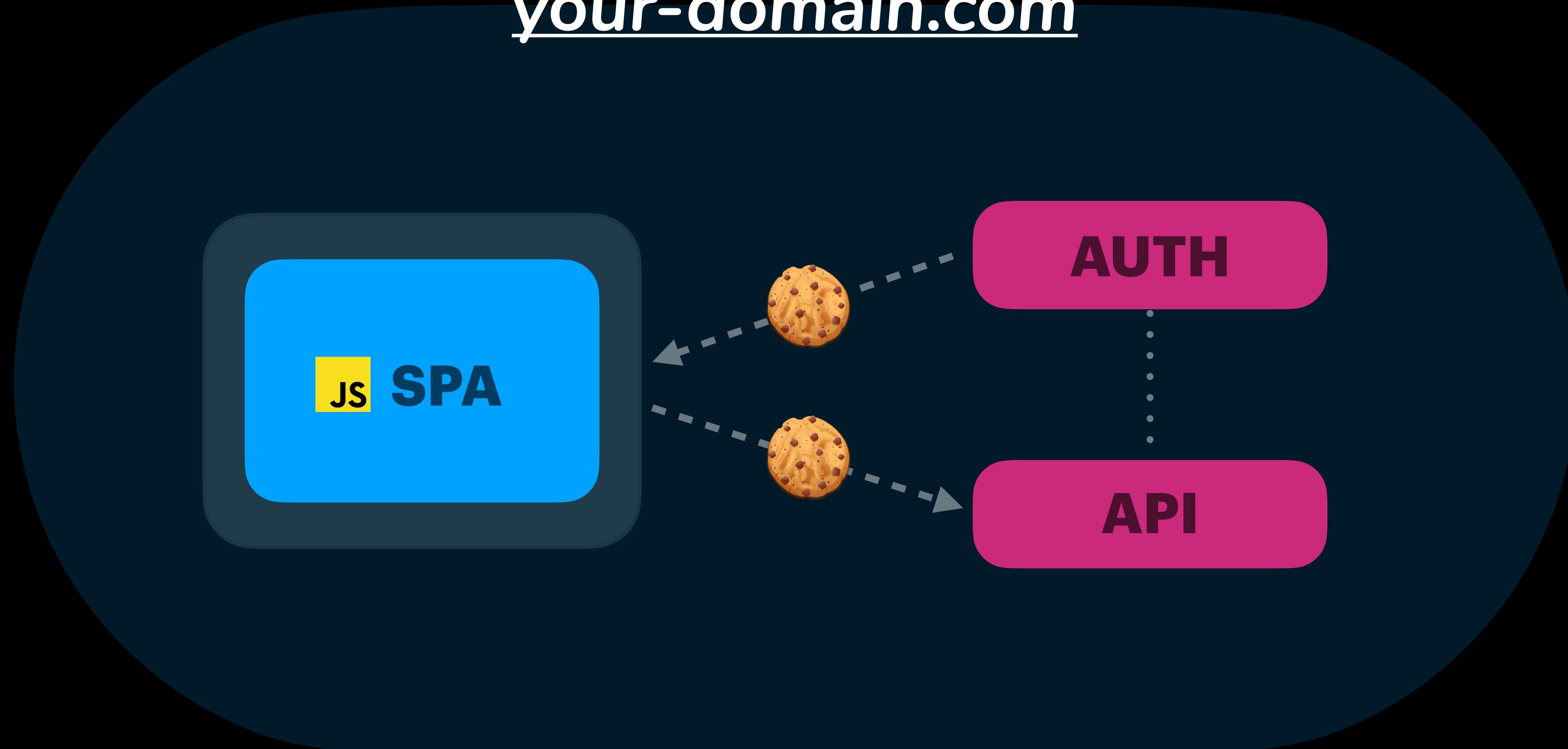
A vertical form with a light blue header bar containing the text "https://SPA". Below this are three dark blue rectangular input fields. The first field contains the text "Email", the second field contains the text "Password", and the third field is empty.

Variante, login e SPA separati

L'unico scopo della SPA è rilevare un errore HTTP e aggiornare la pagina. Il server, non trovando cookie, mostra la schermata di login.



your-domain.com



Cookie-based Authentication

Vantaggi

- Utilizzata da molti anni, le vulnerabilità sono conosciute
- Molto semplice da implementare, trasparente per il front-end (i cookie vengono inviati automaticamente)
- Nessun token da salvare sul front-end (ambiente non sicuro)
- I cookie non vengono letti dal front-end, diminuendo la superficie d'attacco XSS
- I cookie pesano pochissimo (massimo 4kb), poco traffico sulla rete
- Dato che la sessione vive sul server, è facile tracciare e personalizzare lo stato di un utente (es. cambiare i permessi, rimuovere la sessione, ecc...)

Cookie-based Authentication

Svantaggi

- Richiede un back-end (non scontato nel caso di una SPA)
- Richiede che front-end e back-end siano sullo stesso dominio (first-party cookies)
- Serve protezione contro CSRF
- (Non protegge mai completamente contro XSS, le contromisure vanno messe in atto comunque)

Stateful Cookie

Il cookie contiene un puntatore al database che contiene le info sulla sessione

Pro

- Il server può salvare tutto quello che vuole nel database
- Rimuovere una sessione è molto facile

Contro

- Richiede un database
- Aumenta la latenza (si va sul database per ogni chiamata)
- Può essere pesante per il database se si hanno moltissimi utenti

Stateless Cookie

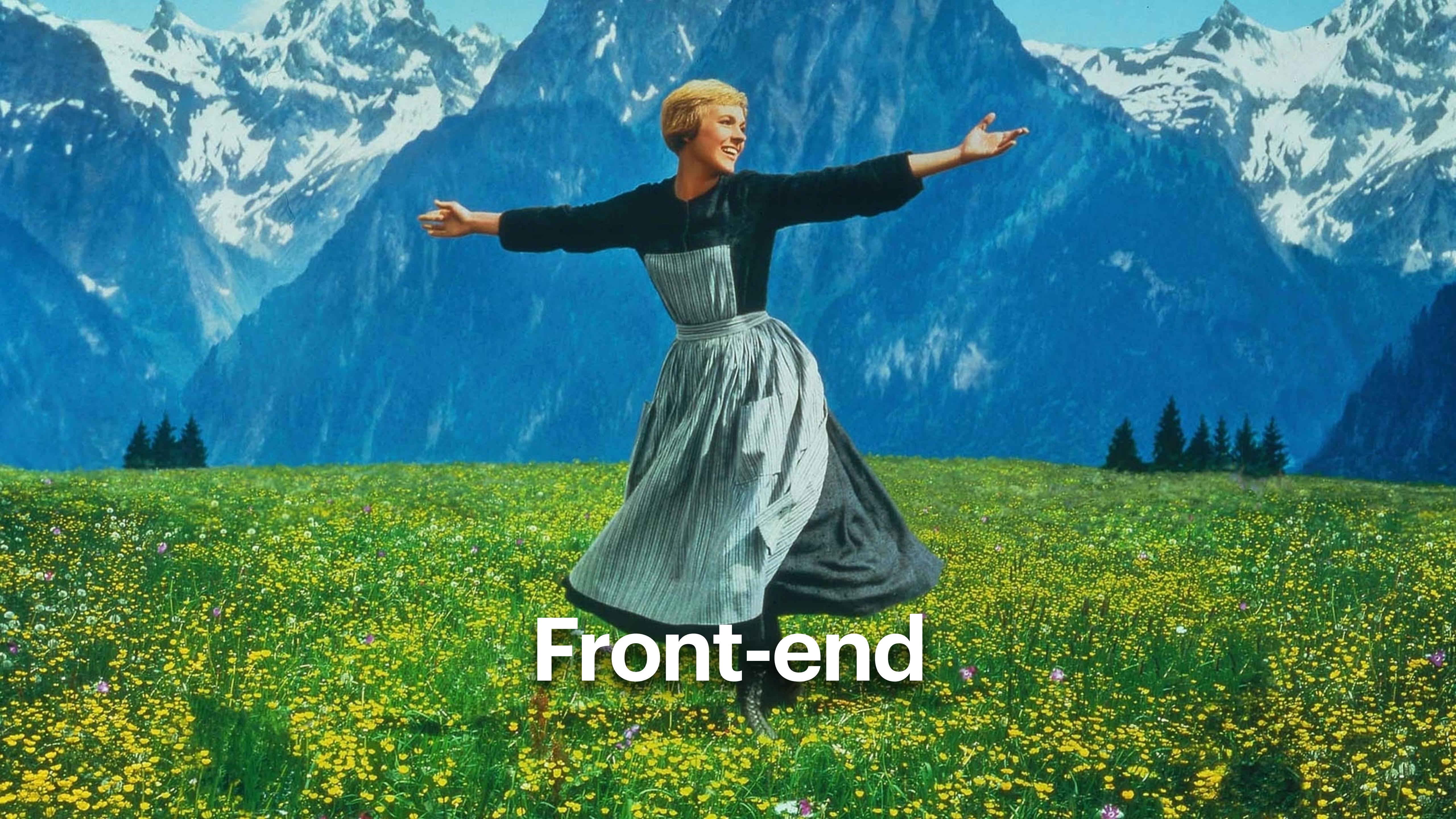
Il cookie contiene info sull'utente (es. l'id), ed è criptato o firmato

Pro

- Riduce la latenza (non serve controllare il database)
- Facile da scalare

Contro

- Il Cookie deve contenere poche informazioni (massimo 4kb)
- Rimuovere una sessione diventa più difficile
- Se usi più server, ogni server deve avere le chiavi per criptare/decriptare il Cookie (o firmarlo)

A classic scene from the movie "The Sound of Music". Maria, played by Julie Andrews, is standing in a lush green field dotted with yellow flowers. She is wearing a traditional green dress with a full, pleated skirt. Her arms are wide open, and she is smiling joyfully. In the background, majestic snow-capped mountains rise against a clear blue sky.

Front-end

Cookie Attributes



Istruzioni per il browser su come trattare un cookie

Cookie Attributes

Expires

Specifica una data di scadenza.

Max-Age

Specifica un tempo massimo di validità. Ha precedenza su *Expires*.

Senza uno di questi due attributi avremmo un [Session Cookie](#), che scade quando l'utente chiude il browser.
In questo caso invece parliamo di [Permanent Cookies](#).

Cookie Attributes

Domain

Il Cookie viene salvato nel browser solo se il dominio coincide.

- Se non lo imposta, viene utilizzato il dominio che l'ha creato, e non permette sotto-domini
- Se lo imposta, i sotto-domini sono consentiti

Cookie Attributes

Path

Il Cookie viene inviato solo se l'URL della richiesta include il path specificato.

Le sotto-cartelle sono incluse, quindi con il path `/api` valgono anche i seguenti:

- `/api/users`
- `/api/users/1`
- `/api/contacts`
- `/api/...`

Cookie Attributes

SameSite

Consente di specificare una strategia per l'invio di Cookie a domini di terze parti.

- **Lax** (Default) - Il Cookie non viene inviato per normali richieste (immagini, iframe, Ajax...) ma viene inviato quando l'utente naviga verso la destinazione (cliccando su un link).
- **Strict** - Il Cookie viene inviato solo dal dominio che l'ha originato.
- **None** - Il Cookie viene sempre inviato. Richiede il flag Secure.

Aiuta a prevenire attacchi di tipo CSRF (Cross-site request forgery).

Cookie Attributes

Secure

Il Cookie viene inviato solo via protocollo HTTPS, ad eccezione di localhost.

Aiuta a prevenire attacchi di tipo **Man in the middle**.

Cookie Attributes

HttpOnly

Il Cookie non può essere letto o modificato da JavaScript nel browser via
document.cookie.

Aiuta a prevenire attacchi di tipo XSS (Cross-site scripting).

Cookie Attributes

Max-Age / Expires

Settano la durata di un cookie

Domain

Il browser lo salva solo se il dominio attuale è quello specificato

Path

Il browser lo invia solo se il path corrisponde

SameSite

Describe se e come un cookie viene inviato in richieste cross-site (fra domini differenti)

Secure

Richiede HTTPS

HttpOnly

Rende il cookie invisibile a JavaScript nel browser

Tipologie di attacchi

- XSS (Cross-site scripting)
- CSRF / XSRF (Cross-site request forgery)
 - Login CSRF

Tipologie di attacchi

Cross-site scripting (XSS)

Iniezione di codice malevolo nel client (browser). L'attaccante riesce a far eseguire del codice sul browser della vittima.

- Esposizione di dati sensibili di un utente
- Furto di credenziali
- Alterazione visiva della pagina web
- Redirect verso siti malevoli o altri comportamenti indesiderati

Esistono 3 tipologie, non discriminanti fra loro: **Persistent**, **Reflected**, **DOM-based**.

Cross-site scripting (XSS)

Persistent

Un sito contiene del codice malevolo indipendentemente dalle azioni dell'utente.
L'attacco ha effetto su tutti gli utenti del sito web, non viene attaccato un utente preciso.

iltuoforum.com

Michele ha scritto:

Ciao mondo! <script>😈</script>

Cross-site scripting (XSS)

Reflected

Un server accetta del codice malevolo da una sorgente esterna (es. link cliccato dall'utente), e genera una pagina web che contiene quel codice.

- Perché Reflected? Il codice viene inviato al server, che poi genera una pagina HTML riflettendo il problema sul client. Più comune nelle Multi-page application.
- Anche chiamato “non-persistent” perché viene attaccato il singolo utente e non tutti gli altri.

<http://iltuosito.com/cerca?termine=<script>😈</script>>

Cross-site scripting (XSS)

DOM-based

Un sito contiene del codice malevolo direttamente nel browser, senza passare per un server.

<http://iltuosito.com/cerca?termine=<script>😈</script>>

Cross-site scripting (XSS)

Come proteggersi? 😬

- Sanitizza l'input dell'utente
 - Utilizza librerie e framework sicuri e con un buon grado di protezione
- Utilizza l'input dell'utente solo quando strettamente necessario
- Evita di tenere informazioni sensibili sul browser
- Configura una Content Security Policy per limitare l'utilizzo di script esterni
- Utilizza i Trusted Types per i browser che li supportano
- Fai particolare attenzione alle librerie che utilizzi
 - Librerie incluse via CDN, link diretto o copy-paste
 - Librerie incluse via package manager (es. NPM)

Tipologie di attacchi

Cross-site request forgery (CSRF)

Ingannare l'utente per fargli inviare una richiesta al sito vulnerabile dopo che si è autenticato su quel sito.



E' un problema utilizzando i Cookie, ma è facilmente risolvibile.

Cross-site request forgery (CSRF)

Esempio 1 (GET)

Un utente viene ingannato, e clicca su un link che fa una chiamata API dannosa.

Quando il browser invia la richiesta, invia anche i Cookie presenti nel browser dell'utente per quel dominio, e se l'utente è autenticato, la richiesta potrebbe andare a buon fine.

1



Da: mariorossi@gmail.com

Ciao Michele! Da quanto tempo. Ti ricordi di me? Ecco una mia foto:

[Foto](http://iltuosito.com/trasferisci?a=mario&somma=1000)

Cross-site request forgery (CSRF)

Esempio 2 (POST)

Un utente viene ingannato, e da un sito malevolo invia un form che fa una richiesta al sito da attaccare.

NB. Il form può essere inviato in automatico senza interazione dell'utente!

```
<body onload="document.forms[0].submit()">  
<form action="iltuosito.com/trasferisci" method="POST">  
  <input type="hidden" name="a" value="Mario" />  
  <input type="hidden" name="somma" value="1000" />  
</form>
```

Cross-site request forgery (CSRF)

Esempio 3 (Login CSRF)

Un utente viene ingannato, e fa una chiamata all'endpoint di autenticazione con le credenziali dell'attaccante!

L'utente, se non è attento, inserirà informazioni sensibili (es. carta di credito) che l'attaccante potrà poi recuperare.

```
<body onload="document.forms[0].submit()">  
  
<form action="iltuosito.com/login" method="POST">  
  <input type="hidden" name="username" value="mario.rossi@gmail.com" />  
  <input type="hidden" name="password" value="1234" />  
</form>
```

Cross-site request forgery (CSRF)

CORS

I browser implementano una strategia chiamata Same-Origin Policy: da un dominio A è impossibile fare chiamate verso un dominio B, per ragioni di sicurezza.

Il server sul dominio B, però, può scegliere di accettare le chiamate provenienti dal dominio A (o da una specifica porta o protocollo), grazie a degli header come Access-Control-Allow-Origin. In questo modo il browser accetta la risposta della chiamata dal dominio B.

Questo meccanismo viene detto Cross-Origin Resource Sharing, e vale per qualsiasi richiesta (anche l'attributo HTML src di un'immagine).

Cross-site request forgery (CSRF)

CORS

Alcune tipologie di chiamate sono dette “Semplici” (GET, HEAD, POST con alcune clausole). Per tutte le chiamate che non lo sono, il browser prima di ogni chiamata cross-domain invia una “Preflight Request” per chiedere al server se è possibile effettuare quella chiamata. Le puoi notare nei Developer Tools del browser con il tipo OPTION.

Se la richiesta Preflight ha successo, la vera chiamata parte.

Penserai che CORS risolva qualsiasi tipo di problema di sicurezza: sbagliato!

Cross-site request forgery (CSRF)

CORS e sicurezza

- Non tutte le chiamate generano una Preflight Request (ad esempio, un semplice form).
- Il meccanismo CORS nasce soprattutto per proteggere il sito che richiede le risorse! È utile solo nel browser, se ad esempio fai una chiamata CURL il CORS non si applica e la chiamata arriva al server in ogni caso.
- In sostanza, CORS non può essere considerato la soluzione per prevenire attacchi CSRF.

Detto questo, è bene usarlo e conoscerlo! Se mai dovessi incontrare un errore di tipo CORS, avvisa gli sviluppatori del back-end.

CSRF Token



Prevenire attacchi di tipo CSRF

Cross-site request forgery (CSRF)

CSRF Token

È un token creato dal server, che viene consegnato al front-end immediatamente (anche prima del login) e che il front-end deve restituire al server per ogni richiesta non-GET. In questo modo il server si assicura che chi sta facendo la richiesta abbia ricevuto il token dal server stesso, perché un attaccante non avrebbe modo di leggere il token dal Cookie da un dominio differente.

Per una pagina web classica, il token può essere inserito direttamente nella pagina HTML generata dal server:

```
<script>  
  const CSRF_TOKEN = "95154d61wec";  
</script>
```

Cross-site request forgery (CSRF)

CSRF Token

È un token creato dal server, che viene consegnato al front-end immediatamente (anche prima del login) e che il front-end deve restituire al server per ogni richiesta non-GET. In questo modo il server si assicura che chi sta facendo la richiesta abbia ricevuto il token dal server stesso, perché un attaccante non avrebbe modo di leggere il token dal Cookie da un dominio differente.

In una Single-Page Application, il token può essere richiesto con una chiamata GET (e letto da un cookie):

```
await fetch("/api/csrf-token");
const CSRF_TOKEN = getCookieByName('CSRF_TOKEN');
```

Cross-site request forgery (CSRF)

CSRF Token

È un token creato dal server, che viene consegnato al front-end immediatamente (anche prima del login) e che il front-end deve restituire al server per ogni richiesta non-GET. In questo modo il server si assicura che chi sta facendo la richiesta abbia ricevuto il token dal server stesso, perché un attaccante non avrebbe modo di leggere il token dal Cookie da un dominio differente.

In ogni caso, il token NON deve essere re-inviato come Cookie. Solitamente lo si invia come header:

```
await fetch("/api/transfer/...", {  
  method: "POST",  
  headers: { "CSRF_TOKEN": "95154d61wec" }  
});
```

CSRF Token

Note

- Esistono più tecniche per i CSRF Token, come il Synchronizer Pattern o la tecnica Double Submit Cookie. In ogni caso il lavoro del Front-End non cambia, la teoria rimane la stessa. In caso di dubbi, confrontatevi con chi sviluppa il back-end per chiarimenti su come ricevere e inviare il token.
- Il CSRF Token può essere ricevuto da un client come Cookie (senza il flag Secure).
- Non inviare mai il CSRF Token come Cookie.

CSRF Token

Non basta il flag *SameSite*?

- SameSite permette i sotto-domini: se sono fidati è ok, altrimenti no
- SameSite non è supportato da browser vecchi (ti interessa Internet Explorer?)
- SameSite non protegge le richieste GET (non dovrebbe essere un problema)
- SameSite non protegge dal caso “Login CSRF” dove un cookie non viene inviato, ma creato



Angular Security

XSS in Angular

Interpolazioni

Il testo interpolato in un template Angular è sempre escaped, quindi un potenziale codice malevolo viene visualizzato ma nulla viene eseguito.

Classe

```
foo = "Hello <script>alert('Boom!')</script>world";
```

Template

```
{{ foo }} // Mostra la stringa "Hello <script>alert('Boom!')</script>world"
```

XSS in Angular

Binding

Il testo in binding in un template Angular viene sanitizzato e il codice potenzialmente malevolo viene rimosso.

Classe

```
foo = "Hello <script>alert('Boom!')</script>world";
```

Template

```
<span [innerHTML]="foo"> // Mostra la stringa "Hello world"
```

XSS in Angular

DomSanitizer

Contiene un metodo per sanitizzare un valore in base a un contesto, e vari metodi per bypassare i controlli di sicurezza di Angular.

```
constructor(sanitizer: DomSanitizer) {  
  this.htmlString = sanitizer.sanitize(SecurityContext.HTML, "<div>...</div>");  
}
```

XSS in Angular

DomSanitizer

Contiene un metodo per sanitizzare un valore in base a un contesto, e vari metodi per bypassare i controlli di sicurezza di Angular.

```
constructor(sanitizer: DomSanitizer) {  
  this.dangerous = 'javascript:alert("hello")';  
  this.trusted = sanitizer.bypassSecurityTrustUrl(this.dangerous);  
}
```

- bypassSecurityTrustHtml
- bypassSecurityTrustScript
- bypassSecurityTrustStyle
- bypassSecurityTrustUrl
- bypassSecurityTrustResourceUrl

CSRF in Angular

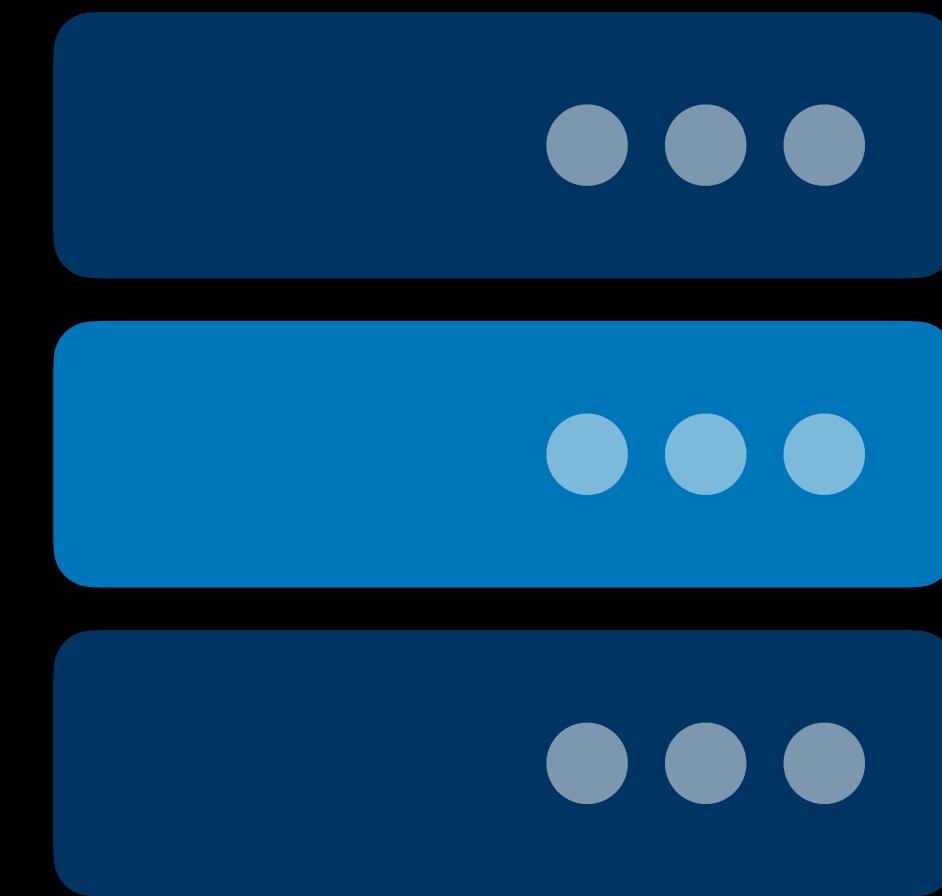
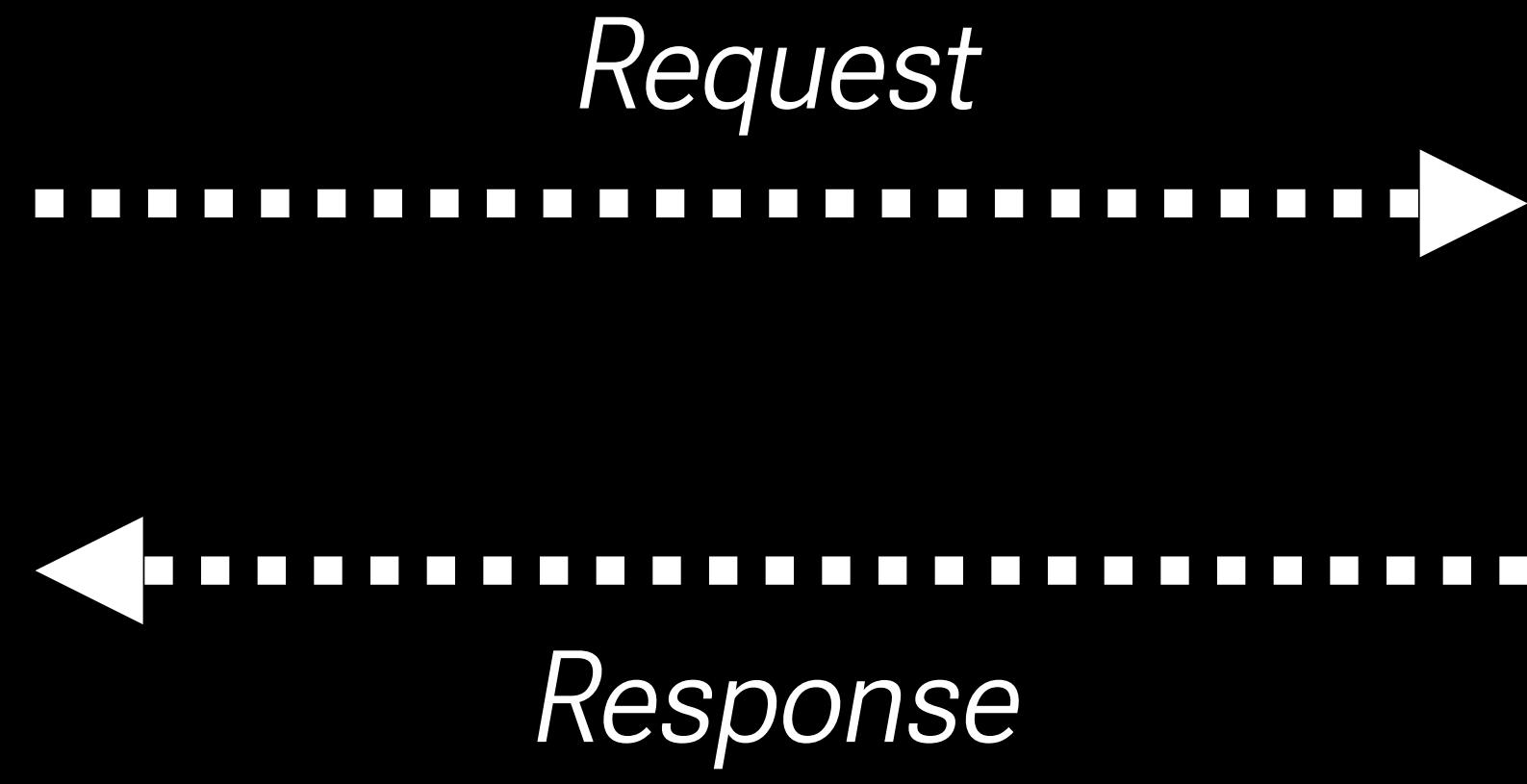
HttpClientXsrfModule

Specifica ad Angular il nome del Cookie che contiene il CSRF Token e il nome dell'header con il quale verrà re-inviato al server ad ogni chiamata. Angular farà il resto!

```
imports: [
  HttpClientModule,
  HttpClientXsrfModule.withOptions({
    cookieName: 'XSRF-TOKEN',
    headerName: 'X-XSRF-TOKEN'
  })
]
```

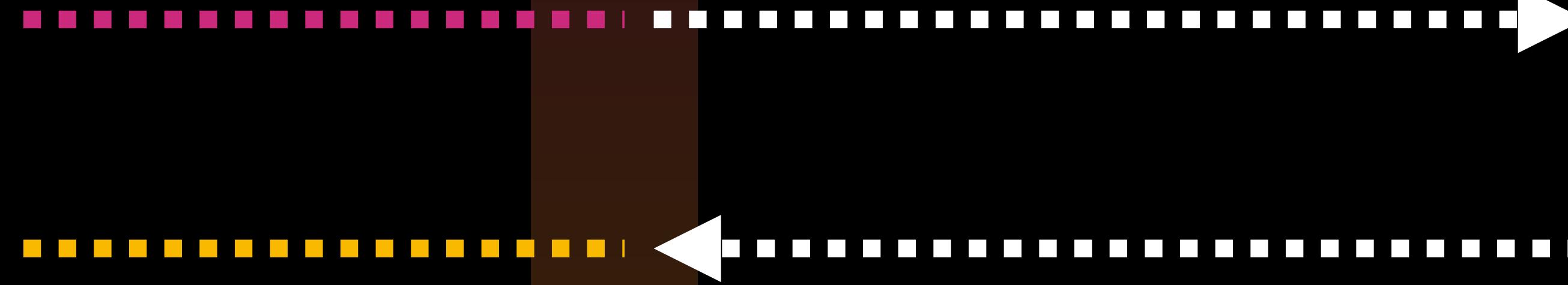


Angular Interceptors





INTERCEPTOR



```
@Injectable()
export class NoopInterceptor implements HttpInterceptor {

  intercept(
    req: HttpRequest<any>,
    next: HttpHandler
  ): Observable<HttpEvent<any>> {
    return next.handle(req);
  }
}
```

Prima della richiesta...

Esempio: modificare la richiesta

```
@Injectable()
export class CookieInterceptor implements HttpInterceptor {

  intercept(
    req: HttpRequest<any>,
    next: HttpHandler
  ): Observable<HttpEvent<any>> {
    const newReq = req.clone({
      withCredentials: true
    })

    return next.handle(newReq);
  }
}
```

...Dopo la risposta

Esempio: intercettare errori

```
@Injectable()
export class ErrorInterceptor implements HttpInterceptor {

  intercept(
    req: HttpRequest<any>,
    next: HttpHandler
  ): Observable<HttpEvent<any>> {
    return next.handle(req).pipe(
      tap({
        error: e => console.error(e)
      })
    );
  }
}
```

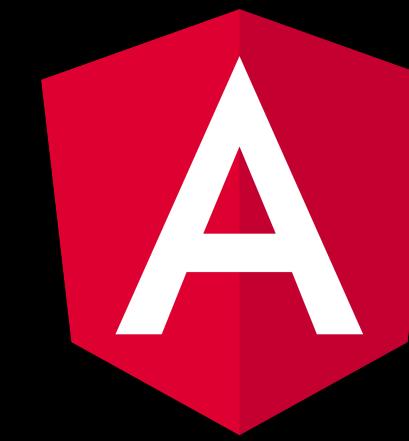
```
import { HTTP_INTERCEPTORS } from '@angular/common/http';

@NgModule({
  providers: [
    {
      provide: HTTP_INTERCEPTORS,
      useClass: CookieInterceptor,
      multi: true
    },
    {
      provide: HTTP_INTERCEPTORS,
      useClass: ErrorInterceptor,
      multi: true
    }
  ]
})
```

Interceptors

Esempi

- Modificare la richiesta prima che parta
- Logout e redirect, se la sessione è scaduta
- Cache globale
- Strategia di retry globale
- Monitorare lo stato delle chiamate (spinner, percentuale...)
- Analytics



Angular Guards

```
RouterModule.forRoot([
{
  path: 'teams',
  loadChildren: () => import('./team.module'),
  canMatch: [TeamsGuard],
  canLoad: [TeamsGuard],
  canActivate: [TeamsGuard],
  canActivateChild: [TeamsGuard],
  canDeactivate: [TeamsGuard]
},
{
  path: '**',
  component: NotFoundComponent
}
])
```

Modulo da caricare (o componente, con *loadComponent*)

```
RouterModule.forRoot([
{
  path: 'teams',
  loadChildren: () => import('./team.module'),
  canMatch: [TeamsGuard],
  canLoad: [TeamsGuard],
  canActivate: [TeamsGuard],
  canActivateChild: [TeamsGuard],
  canDeactivate: [TeamsGuard]
},
{
  path: '**',
  component: NotFoundComponent
}
])
```

*Can this route be
matched?*

```
RouterModule.forRoot([
{
  path: 'teams',
  loadChildren: () => import('./team.module'),
  canMatch: [TeamsGuard],
  canLoad: [TeamsGuard],
  canActivate: [TeamsGuard],
  canActivateChild: [TeamsGuard],
  canDeactivate: [TeamsGuard]
},
{
  path: '**',
  component: NotFoundComponent
}
])
```

Can the file be loaded?

```
RouterModule.forRoot([
{
  path: 'teams',
  loadChildren: () => import('./team.module'),
  canMatch: [TeamsGuard],
  canLoad: [TeamsGuard],
  canActivate: [TeamsGuard],
  canActivateChild: [TeamsGuard],
  canDeactivate: [TeamsGuard]
},
{
  path: '**',
  component: NotFoundComponent
}
])
```



Can this route be activated?

```
RouterModule.forRoot([
{
  path: 'teams',
  loadChildren: () => import('./team.module'),
  canMatch: [TeamsGuard],
  canLoad: [TeamsGuard],
  canActivate: [TeamsGuard],
  canActivateChild: [TeamsGuard],
  canDeactivate: [TeamsGuard]
},
{
  path: '**',
  component: NotFoundComponent
}
])
```



Can children routes be activated?

```
RouterModule.forRoot([
{
  path: 'teams',
  loadChildren: () => import('./team.module'),
  canMatch: [TeamsGuard],
  canLoad: [TeamsGuard],
  canActivate: [TeamsGuard],
  canActivateChild: [TeamsGuard]
  canDeactivate: [TeamsGuard]
},
{
  path: '**',
  component: NotFoundComponent
}
])
```



*Can this route be
deactivated?*

```
@Injectable()
class TeamsGuard implements CanMatch, CanLoad, CanActivate, CanActivateChild, CanDeactivate {
```



```
    canMatch(
        route: ActivatedRouteSnapshot,
        segments: UrlSegment[]
    ): Observable<boolean|UrlTree>|Promise<boolean|UrlTree>|boolean|UrlTree {
        return ...;
    }

    canLoad(route: Route, segments: UrlSegment[]) {
        return ...;
    }

    canActivate(route: ActivatedRouteSnapshot, state: RouterStateSnapshot) {
        return ...;
    }

    canActivateChild(childRoute: ActivatedRouteSnapshot, state: RouterStateSnapshot) {
        return ...;
    }

    canDeactivate(
```

```
        }

        canActivate(route: ActivatedRouteSnapshot, state: RouterStateSnapshot) {
            return ...;
        }

        canActivateChild(childRoute: ActivatedRouteSnapshot, state: RouterStateSnapshot) {
            return ...;
        }

        canDeactivate(
            component: TeamsComponent,
            currentRoute: ActivatedRouteSnapshot,
            currentState: RouterStateSnapshot,
            nextState: RouterStateSnapshot
        ) {
            return !component.hasUnsavedChanges();
        }
    }
}
```

Guard Functions

```
RouterModule.forRoot([
  {
    path: 'teams',
    loadChildren: () => import ('./team.module'),
    canMatch: [(route, segments) => true],
    canLoad: [(route, segments) => true],
    canActivate: [(route, state) => true],
    canActivateChild: [(childRoute, state) => true],
    canDeactivate: [(component, route, state, nextState) => !component.hasUnsavedChanges()]
  },
  {
    path: '**',
    component: NotFoundComponent
  }
])
```

Guards

WITH DEPENDENCIES

```
@Injectable()  
export class TeamsGuard implements CanMatch {  
  
  constructor(private service: PermissionsService) {}  
  
  canMatch() {  
    return this.service.isTeamManager();  
  }  
}
```

Guard Functions

WITH DEPENDENCIES

```
import { inject } from '@angular/core';

RouterModule.forRoot([
  {
    path: 'teams',
    loadChildren: () => import('./team.module'),
    canMatch: [() => inject(PermissionsService).isTeamManager()]
  },
  {
    path: '**',
    component: NotFoundComponent
  }
])
```

Same Path

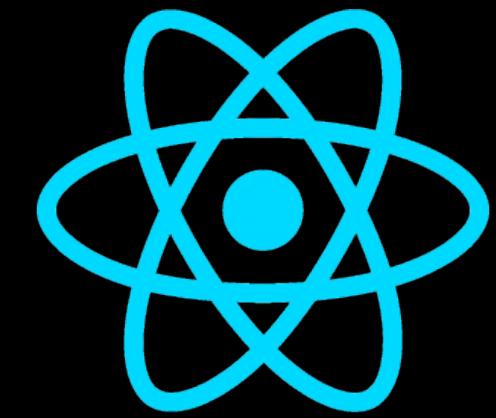
With *canMatch*

```
{  
  path: 'teams',  
  component: NewTeamsComponent,  
  canMatch: [() => inject(PermissionsService).hasPermission('v2')]  
,  
{  
  path: 'teams',  
  component: TeamsComponent  
}
```

Guards != Security

Una guardia **NON** è una misura di sicurezza! Ciò che è nel browser può sempre essere visto dagli 😎 hacker.

Un'applicazione è sicura quando il **server** è sicuro.



React Security

Interpolazioni

Le espressioni in un template JSX sono escaped, quindi un potenziale codice malevolo viene visualizzato ma nulla viene eseguito.

```
const title = "Hello <script>alert('Boom!')</script>world";
// Mostra la stringa"Hello <script>alert('Boom!')</script>world"
const element = <h1>{title}</h1>
```

XSS in React

innerHTML

Per inserire del codice HTML nella pagina, al posto del classico `innerHTML` va utilizzata la proprietà `dangerouslySetInnerHTML`, ma solo se siamo certi del contenuto.

```
const content = "Hello <script>alert('Boom!')</script>world";  
  
// L'alert viene eseguito  
<h1 dangerouslySetInnerHTML={ __html: content } >
```

XSS in React

React non controlla le proprietà

Assicurati di utilizzare contenuti sicuri quando li passi come proprietà: non sono escaped!
Vale anche per le prop dei tuoi componenti.

```
const href = "javascript:alert('Hacked!')";
```

```
// L'alert viene eseguito  
<a href={href}>Link</a>
```

Proprietà dinamiche

Attenzione quando permetti a un contenuto esterno di impostare delle prop per te!

```
// Immagina che arrivino da una chiamata HTTP
const customProps = {
  dangerouslySetInnerHTML: { __html: '' }
}
// Problema!
<div {...customProps} />
```

React non sanifica

“Sanificare” (*sanitize*) vuol dire rimuovere potenziali pericoli da un codice. Ad esempio, rimuovere dei tag `<script>` da una stringa. React non fornisce tool per farlo, ma se dovesse servire, puoi usare librerie come **DOMPurify**.

```
const content = "Hello <script>alert('Boom!')</script>world";
// Lo script viene rimosso
const sanitized = DOMPurify.sanitize(content);

<h1 dangerouslySetInnerHTML={ _html: sanitized } />
```

Authorization Servers



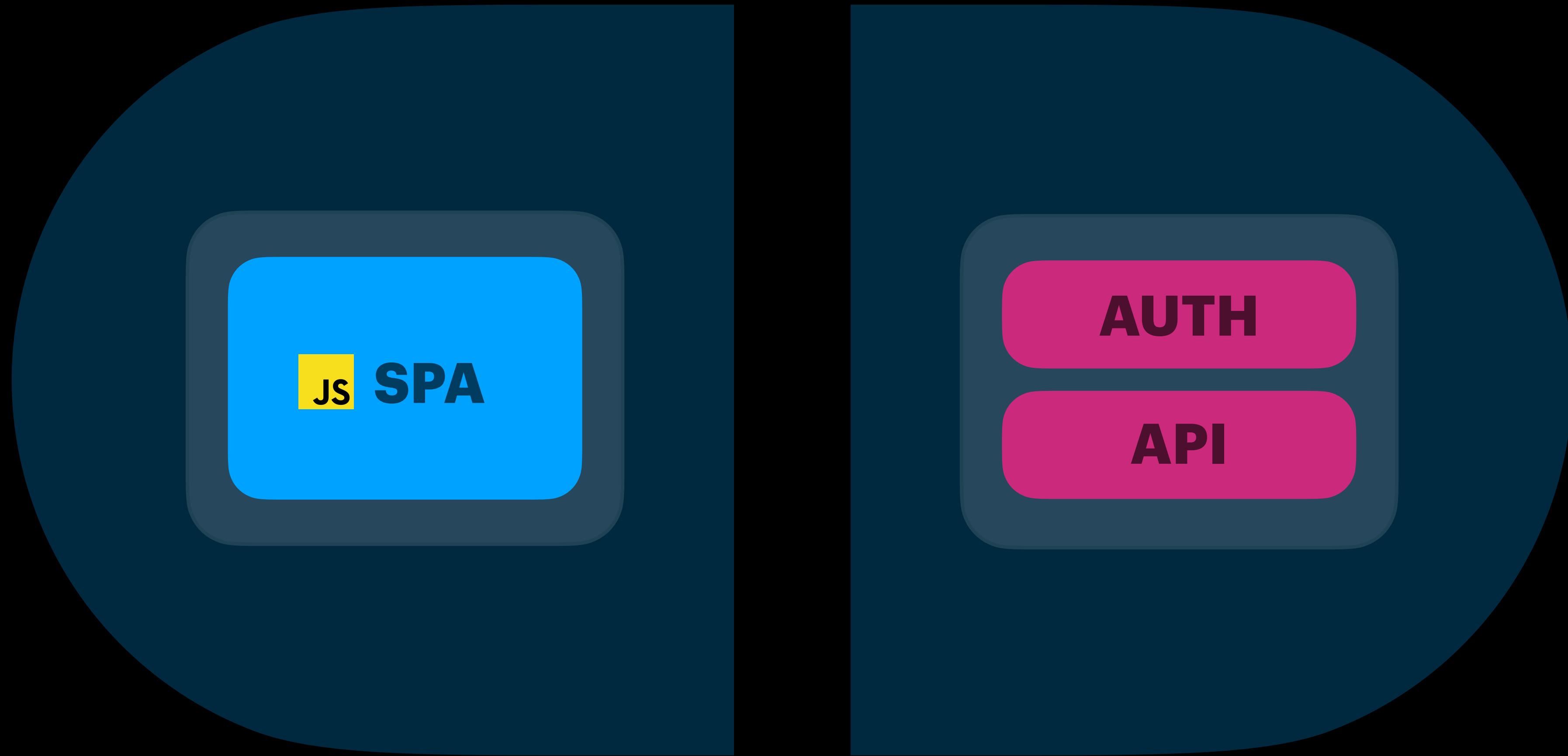
Introduzione a OAuth

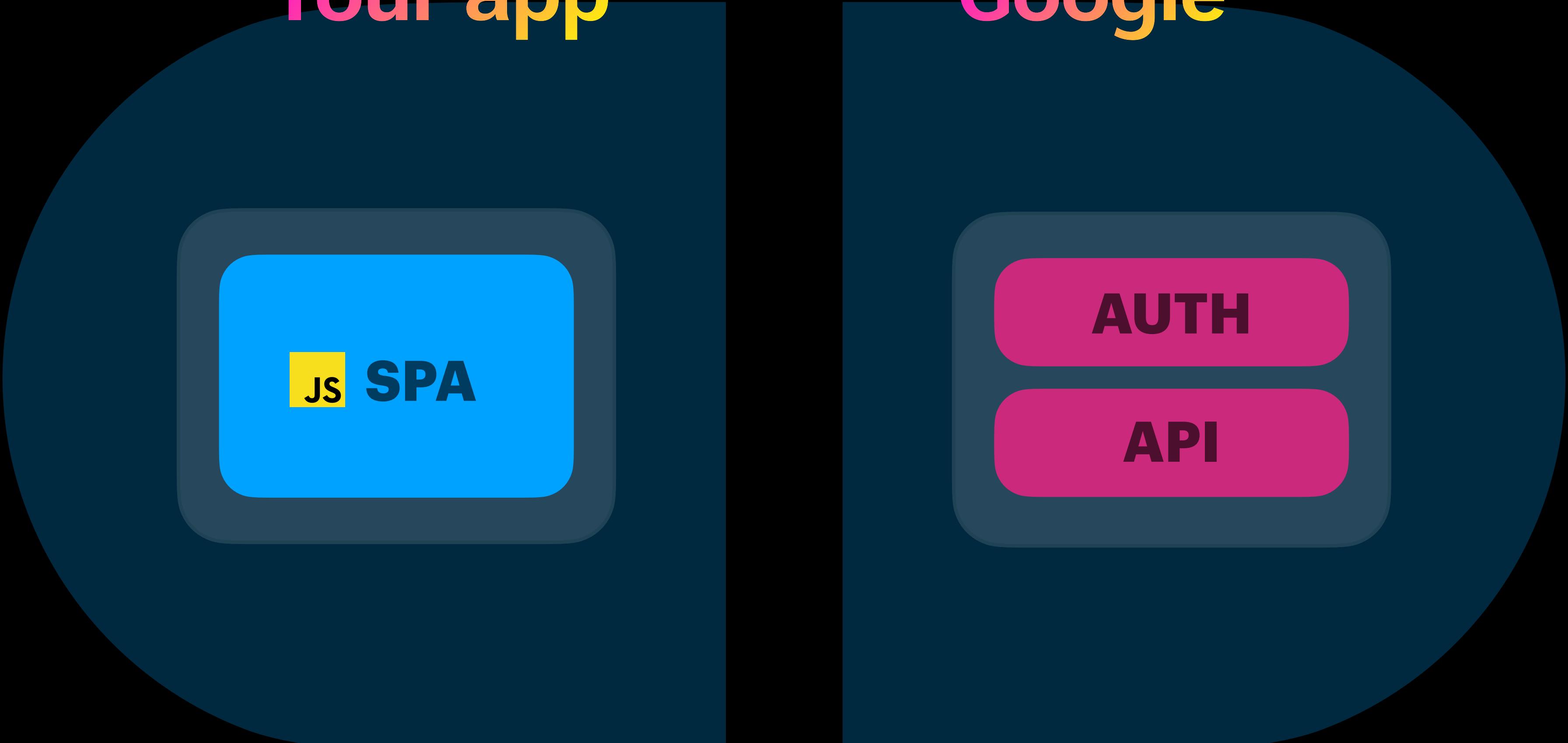
your-domain.com

JS SPA

AUTH

API





Your app

JS SPA

Google

AUTH
API

In un mondo ideale...



spotify.com

Entra con le tue credenziali di Google!

Email

Password

Entra



spotify.com

Entra con le tue credenziali di Google!

NO, GRAZIE

Email

Password

Entra

Single Sign-on

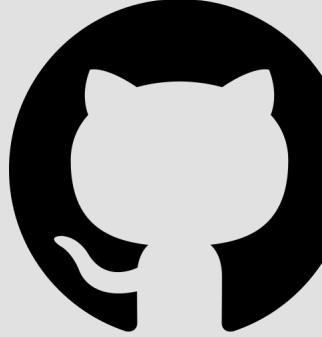
Un'unica identità per più applicazioni



Login con Google



Login con Facebook



Login con Github

Client

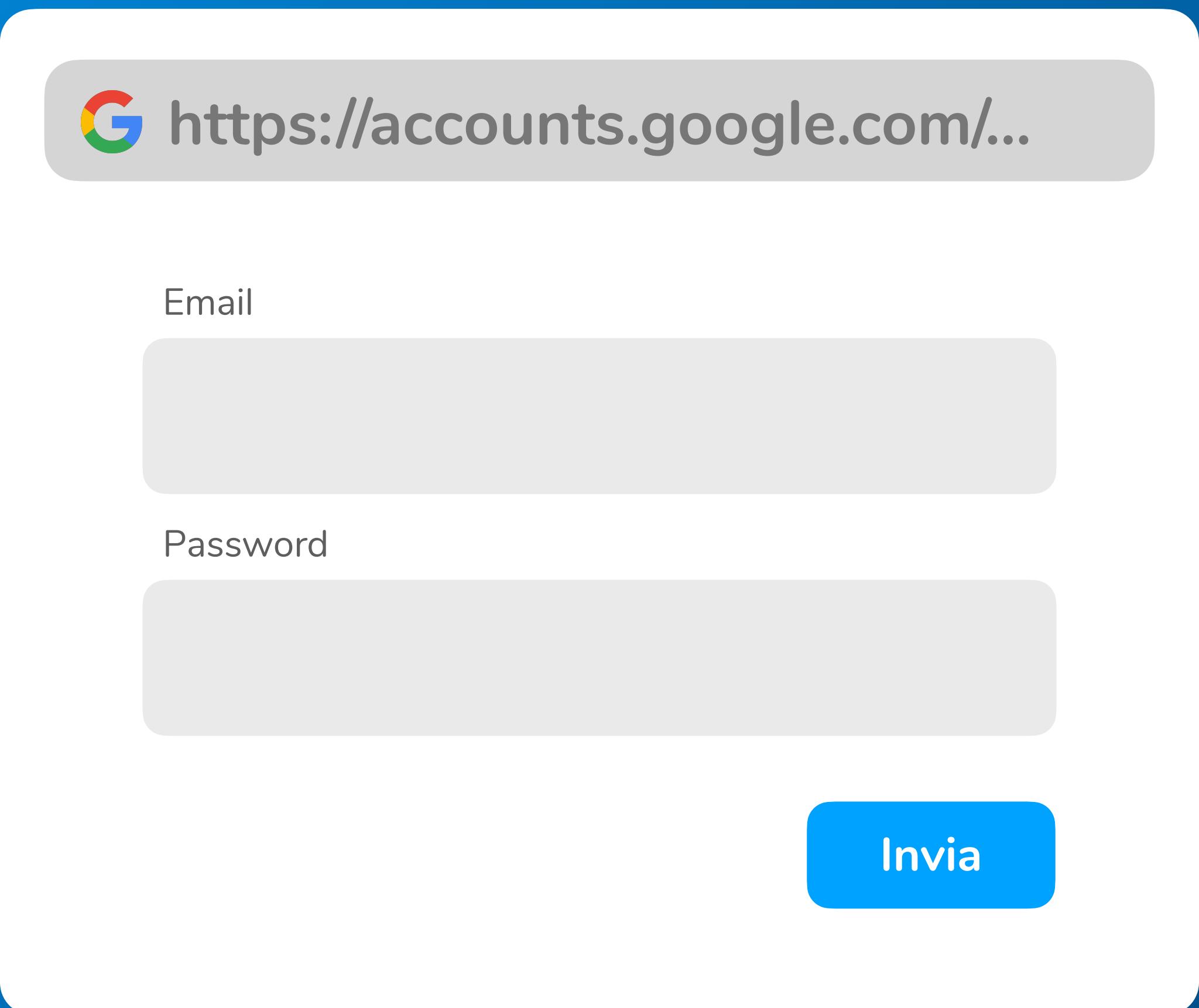
`https://SPA`

Non sei loggato!



Login con Google

Authorization Server



A placeholder image of a Google account sign-in page. It features a header bar with the Google logo and the URL `https://accounts.google.com/...`. Below the header are two large, light-gray rectangular input fields labeled "Email" and "Password". At the bottom right is a blue button with the white text "Invia".

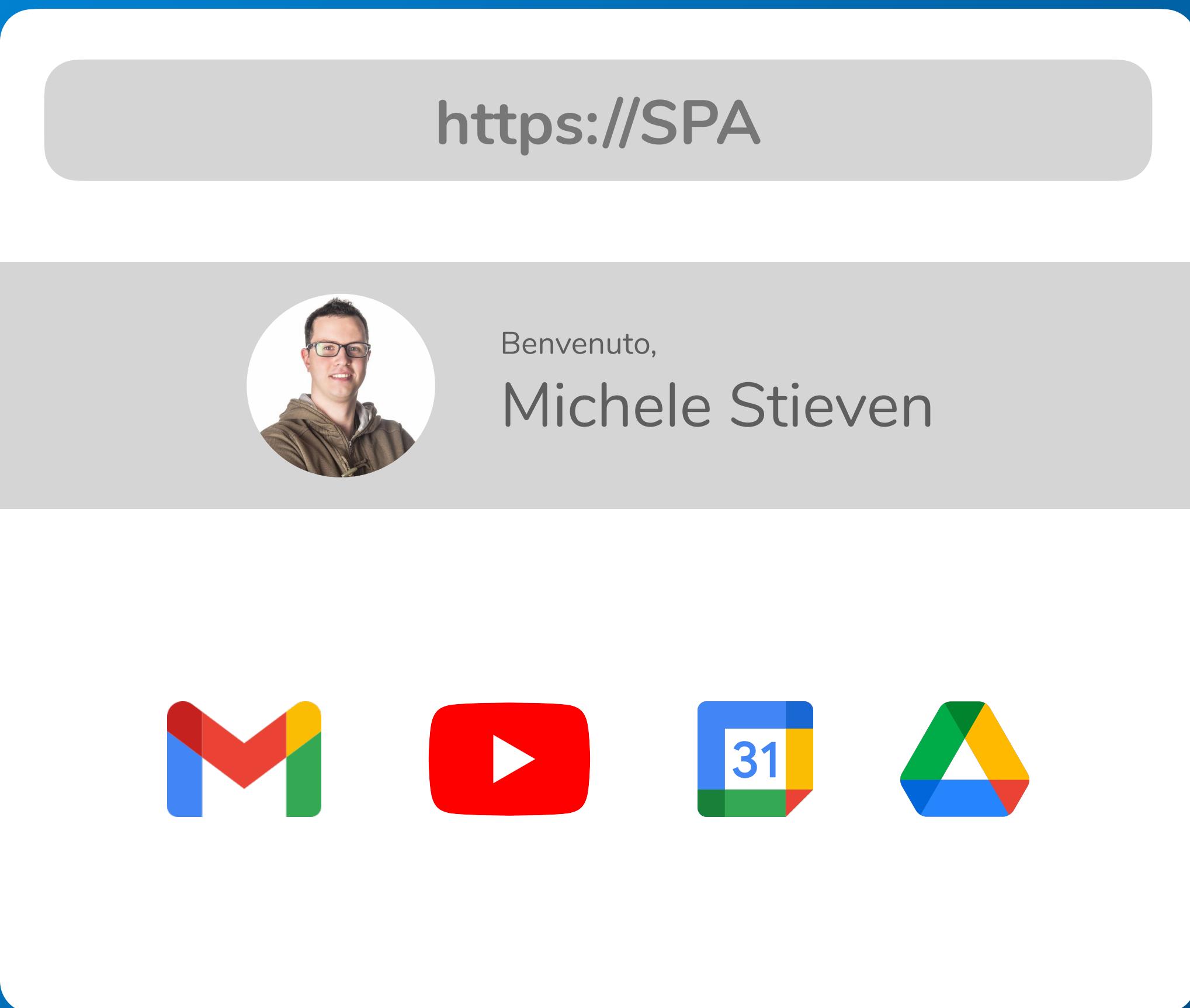
https://accounts.google.com/...

Email

Password

Invia

Client



Configurazione del progetto



[https://google.com/...](https://google.com/)

Nome dell'app

MyProject

URL dell'app

myproject.com

Redirect URL

myproject.com/auth-callback

Client ID

dfhge7e834efn92e102wch

Client Secret

4389oiruhj3039fdc1g2e9d249rf

PUBLIC



PRIVATE

Client

<https://myproject.com>

Non sei loggato!



Login con Google

Authorization Server

 [https://accounts.google.com/...](https://accounts.google.com/)

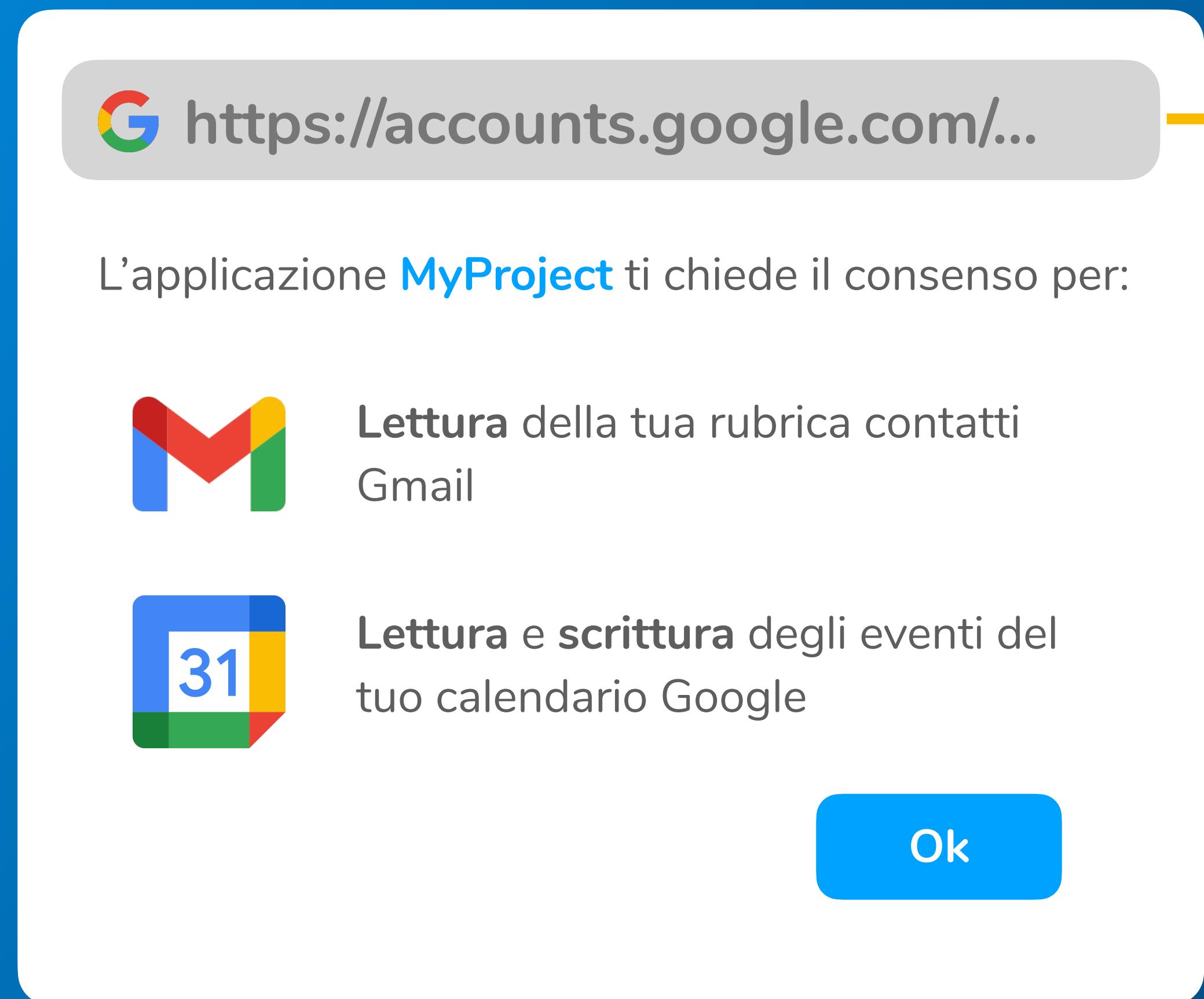
Email

Password

Invia

?client_id=...
?redirect_uri=...
...
?scopes=...

Authorization Server



?client_id=...
?redirect_uri=...
...
?scopes=...

Client

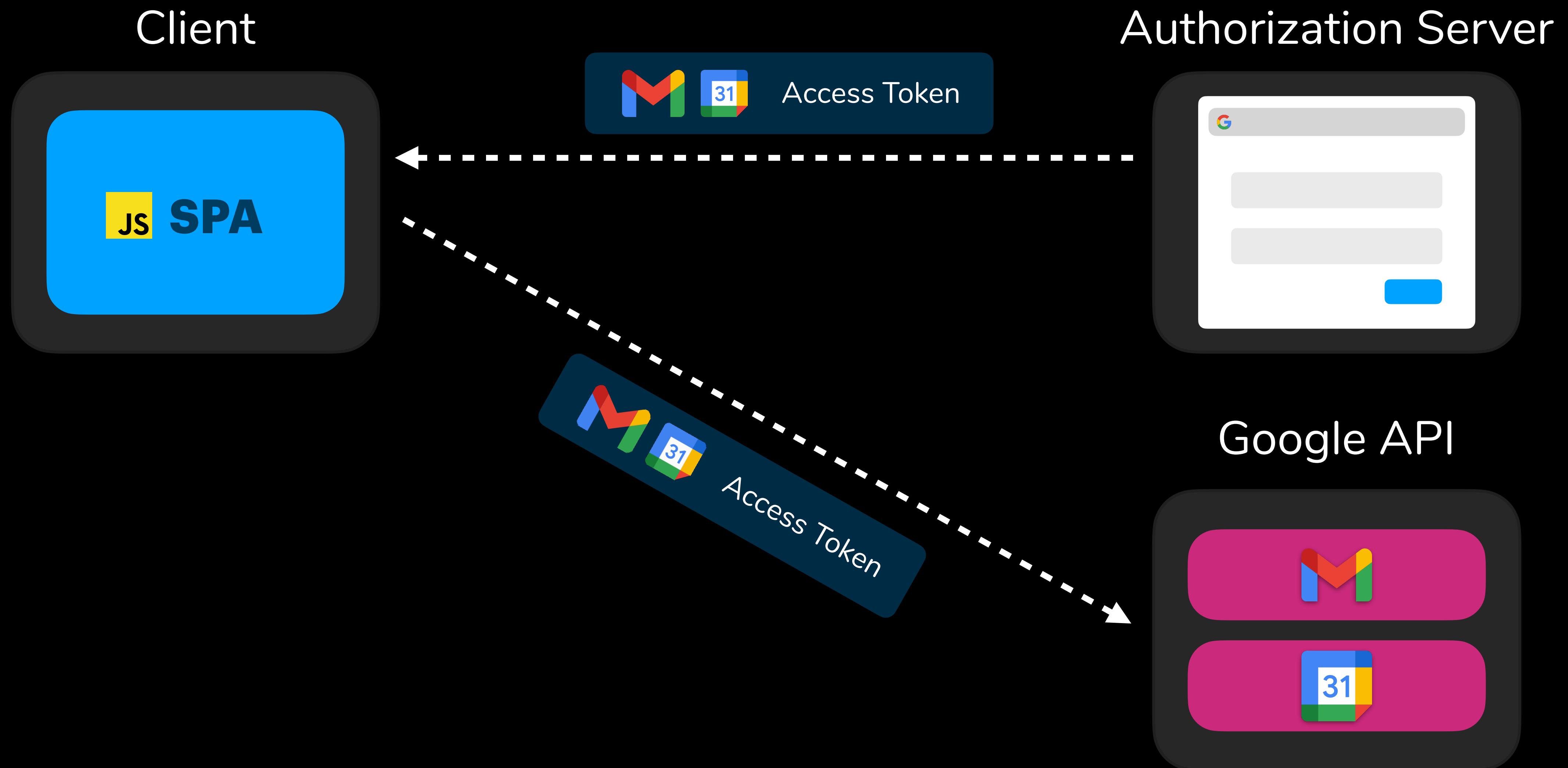
<https://myproject.com/auth-callback>



Benvenuto,
Michele Stieven



Access Token
verview3r92930r1eg0asjckv0rgjv





OAuth 2.0

OAuth 2.0 is the industry-standard [protocol for authorization](#).
OAuth 2.0 focuses on client developer simplicity while providing
specific [authorization flows](#) for web applications, desktop
applications, mobile phones, and living room devices.

OAuth

Actors

- **Client** - L'applicazione che vuole usare le risorse dell'utente
- **Resource Owner** - Il proprietario della risorsa (l'utente)
- **Resource Server** - Il server che contiene la risorsa (es. Google API)
- **Authorization Server** - Il server che rilascia i token per accedere alle risorse sul Resource Server

Clients

- **Client confidenziali**
 - Possono tenere un segreto [client secret]
 - Esempi: un server, un sito web con un back-end
- **Client pubblici**
 - Non possono tenere un segreto
 - Esempi: una SPA senza back-end, un'app mobile, un'app desktop

OAuth 2.0 offre varie tecniche di autenticazione fra cui scegliere, chiamate Flow. È importante scegliere quella adatta in base alla tipologia di Client!

Scope

Lista di permessi che il Client richiede per conto dell’utente. L’utente, quando si autentica, deve acconsentire ai permessi per permettere all’app di funzionare correttamente.

Non è detto che l’utente debba consentire tutti i permessi: alcuni Authorization Server permettono all’utente di “togliere la spunta” dagli scope che non vuole consentire (es. Facebook), perdendo magari parte delle funzionalità dell’applicazione.

Access Token

Un Access Token è un **lasciapassare** che un Authorization Server consegna al Client per accedere alle risorse dell'utente sul Resource Server.

OAuth non specifica in che formato dev'essere il token, anche se solitamente si utilizzano i JWT.

Questo token può anche essere criptato e reso illeggibile al Client, che non sa cosa contiene: la cosa importante è consegnarlo al server ad ogni richiesta, che lo validerà ogni volta.

OAuth

Refresh Token

Un token che può essere usato per richiedere un nuovo Access Token quando quest'ultimo scade.

Come livello di pericolosità, si avvicina molto a delle credenziali. Può durare molto a lungo, ma può anche essere revocato. È ideale tenerlo in un ambiente sicuro, non in un Client Pubblico.

Comparazione

Access Token

- Scade dopo poco (minuti, ore)
- Può essere consegnato a un Client Pubblico
- Permette l'accesso a delle risorse
- Solitamente non può essere revocato

Refresh Token

- Può durare a lungo (giorni, mesi)
- Può essere consegnato a un Client Confidenziale
- Permette di ricevere un nuovo Access Token
- Deve poter essere revocato

OAuth

Flow/Grant

OAuth specifica più modi per ottenere un Access Token. Questi vengono detti **Flow**. Eccone alcuni:

- Implicit Flow (per client pubblici) [sconsigliato]
- Authorization Code Flow (per client confidenziali)
- Authorization Code Flow with PKCE (per client pubblici)
- Client Credential Flow (server-to-server, nessun utente specifico)
- Resource Owner Password Flow [legacy, sconsigliato]
- Device Flow (CLI, Televisioni, Decoder...)

Implicit Flow



Per client pubblici... ma obsoleto

OAuth Flows

Implicit Flow

Client pubblici

Sconsigliato

Il Client richiede un Access Token all'Authorization Server, e quest'ultimo glielo restituisce direttamente.

Il token viene passato via URL, per questo è sconsigliato (può rimanere in cronologia ed essere intercettato da estensioni).

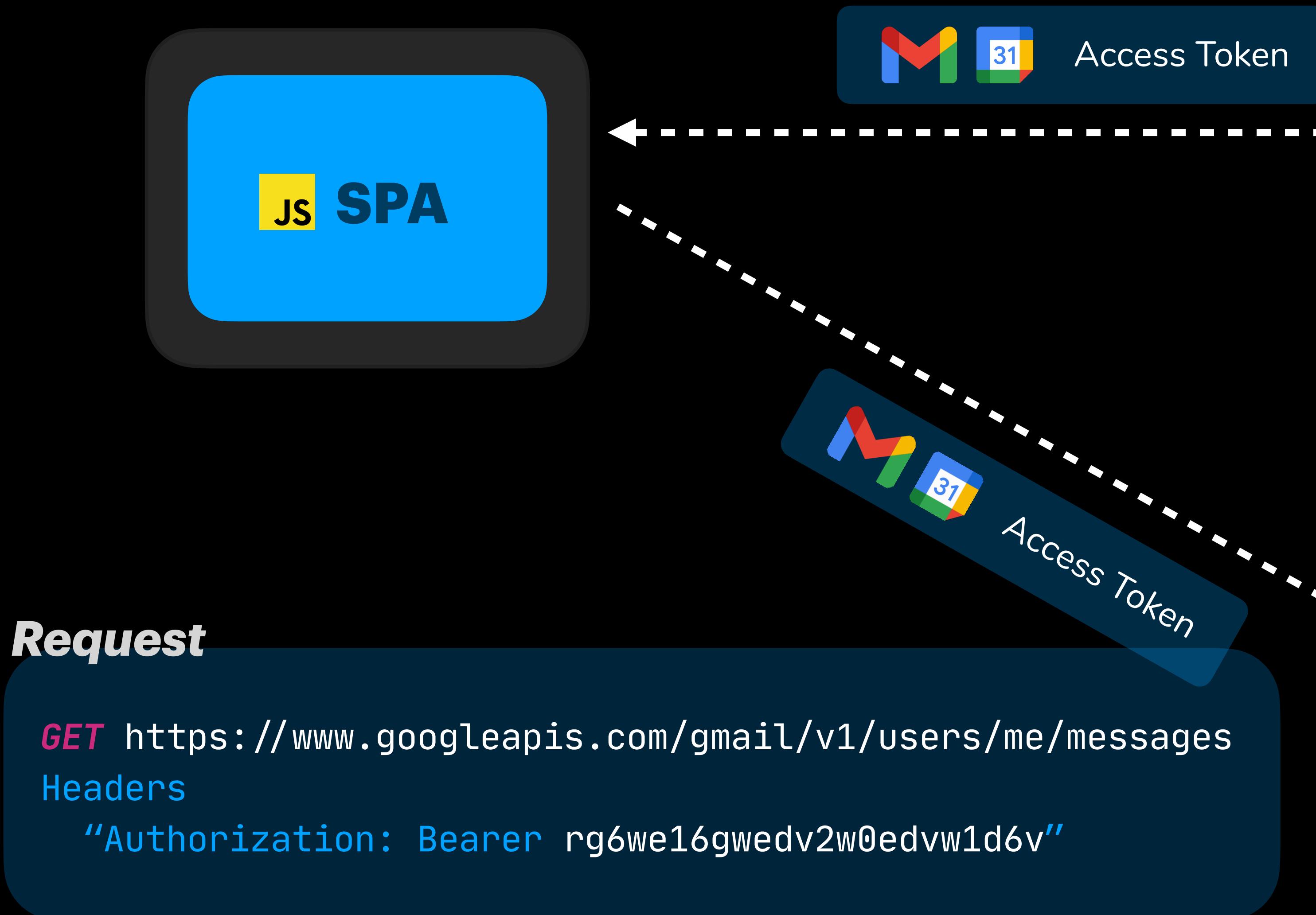
Request

```
GET https://accounts.google.com/o/oauth2/auth  
?scope=gmail.insert gmail.send  
&redirect_uri=https://your-app.com/callback  
&response_type=token  
&client_id=86151612616  
&state=wef5w6ed1w6w
```

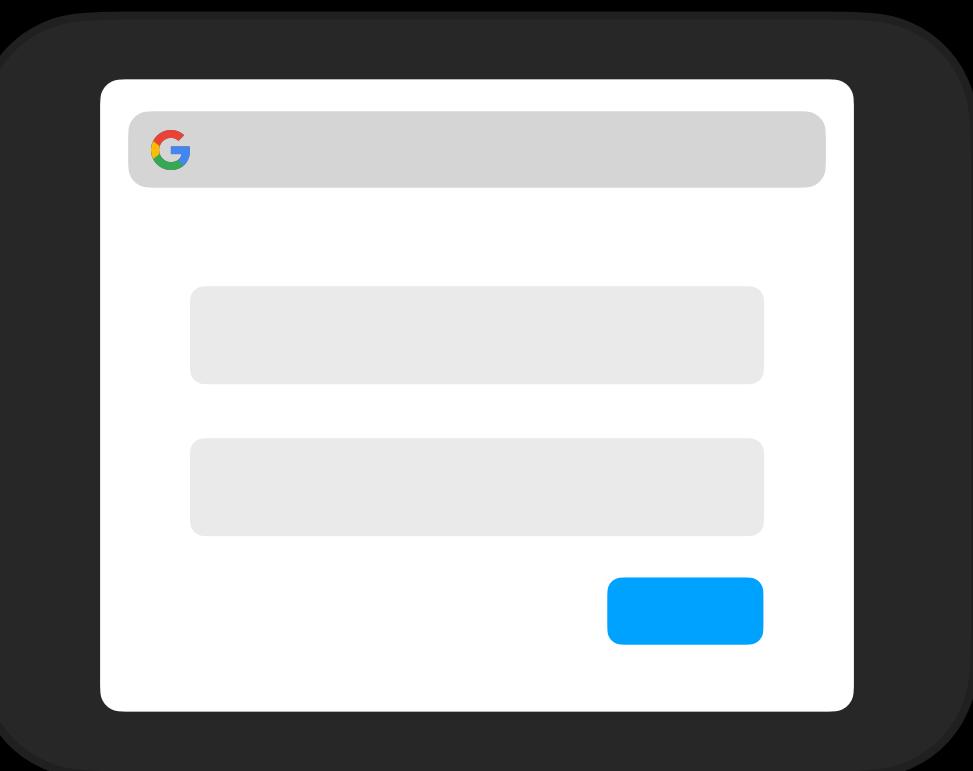
Response

```
https://your-app.com/callback  
#access_token=rg6we16gwedv2w0edvw1d6v  
&token_type=Bearer  
&expires_in=600  
&state=wef5w6ed1w6w
```

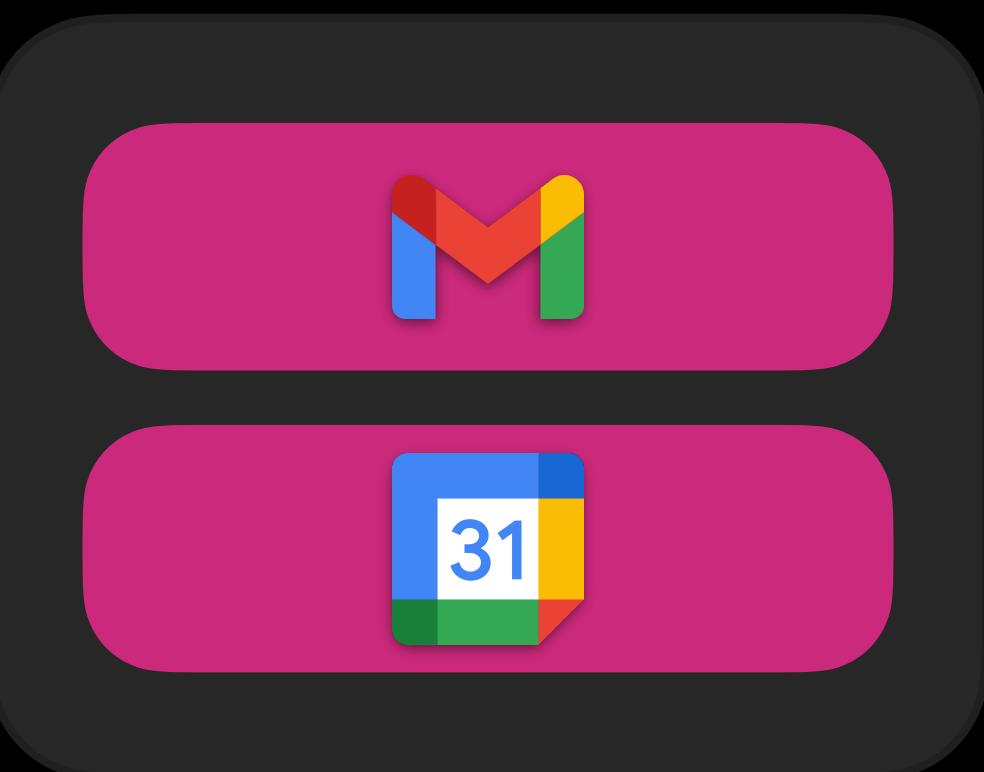
Client



Authorization Server



Google API



OAuth Flows

Silent Refresh

Client pubblici

Obsoleto

L'Implicit Flow non restituisce un Refresh Token.

Questa tecnica è un'alternativa per il rinnovo
dell'Access Token.

Viene creato un iframe nascosto per fare una
richiesta di autorizzazione con il parametro
`prompt=none`. È una specifica di OpenID
Connect.

Request

```
GET https://accounts.google.com/o/oauth2/auth  
?scope=openid...  
&redirect_uri=https://your-app.com/callback  
&response_type=token  
&client_id=86151612616  
&state=wef5w6ed1w6w  
&prompt=none
```

È un approccio destinato a morire,
perché i browser stanno lentamente
soppiantando i Cookie di terze parti!

Authorization Code Flow



La punta di diamante di OAuth, per client confidenziali

OAuth Flows

Authorization

Code Flow

Client confidenziali

Il Client richiede un Codice all'Authorization Server, quest'ultimo lo restituisce e il back-end scambia questo Codice per un Access Token.
L'Access Token rimane sul server.

Request

```
GET https://accounts.google.com/o/oauth2/auth  
?scope=gmail.insert gmail.send  
&redirect_uri=https://your-backend.com/callback  
&response_type=code  
&client_id=86151612616  
&state=wef5w6ed1w6w
```

Response

```
https://your-backend.com/callback  
?code=rg6we16gwedv2w0edvw1d6v
```

Request

```
POST https://oauth2.googleapis.com/token  
code=rg6we16gwedv2w0edvw1d6v  
client_id=86151612616  
client_secret=efwe5f6w1e2f1we6f  
redirect_uri=https://your-backend/token  
grant_type=authorization_code
```

OAuth Flows

Authorization

Code Flow

Client confidenziali

Non è detto che il server ritorni un Refresh Token.

Spesso, per richiederlo, va specificato lo scope
offline_access.

Final Response

```
{  
  "access_token": "wefwe61fwvs2",  
  "token_type": "Bearer",  
  "expires_in": 3600,  
  "refresh_token": "dstgn541sns61g"  
}
```

OAuth Flows

Authorization Code Flow

Client confidenziali

Puoi utilizzare il Refresh Token per ottenere un nuovo Access Token tramite l'endpoint /token.

Soltamente si ottiene anche un nuovo Refresh Token (*Rotation*).

Request

```
POST https://oauth2.googleapis.com/token  
client_id=86151612616  
client_secret=efwe5f6w1e2f1we6f  
refresh_token=dstgn541sns61g  
grant_type=refresh_token
```

OAuth Flows

Authorization

Code Flow

Client confidenziali

Il Refresh Token può essere revocato in qualsiasi momento (logout dell'utente, l'utente toglie i permessi dall'app, scadenza...), in quel caso l'utente deve loggarsi nuovamente.

OAuth - Authorization Code Flow

Back-end for Front-end (BFF)

Il flow più sicuro, quando è coinvolto un utente, è l'Authorization Code. Una Single-Page Application, però, non è un Client confidenziale, e non può tenere un segreto.

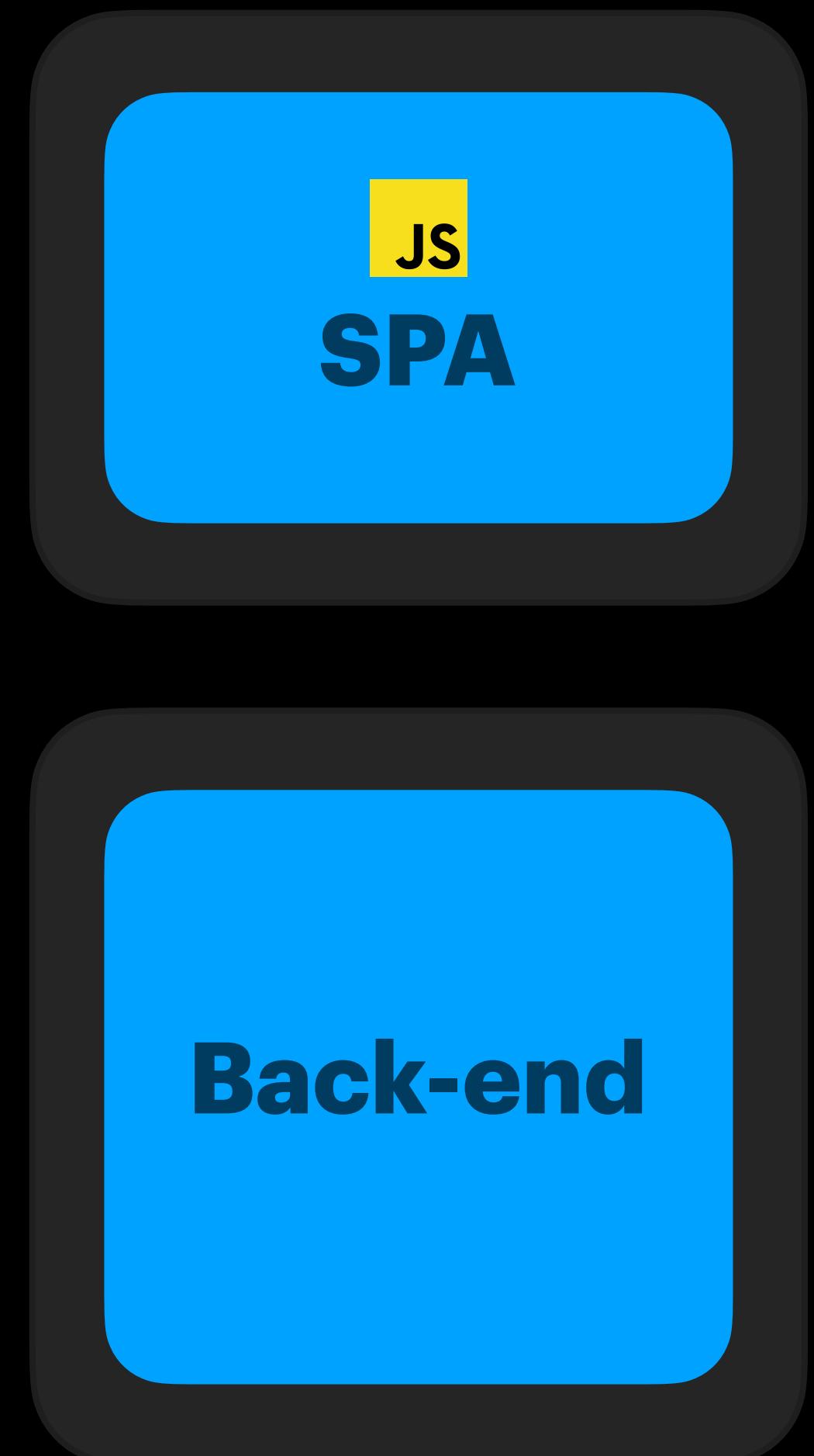
L'architettura BFF è un buon compromesso: se la SPA ha un back-end sullo stesso dominio, puoi usare quello per ricevere i vari Token, che verranno salvati in memoria o in database per l'utente. Il back-end terrà l'utente autenticato sul front-end grazie a un semplice Cookie.

Il front-end non vede mai i Token, e per ogni chiamata al Resource Server dovrà chiedere al back-end.



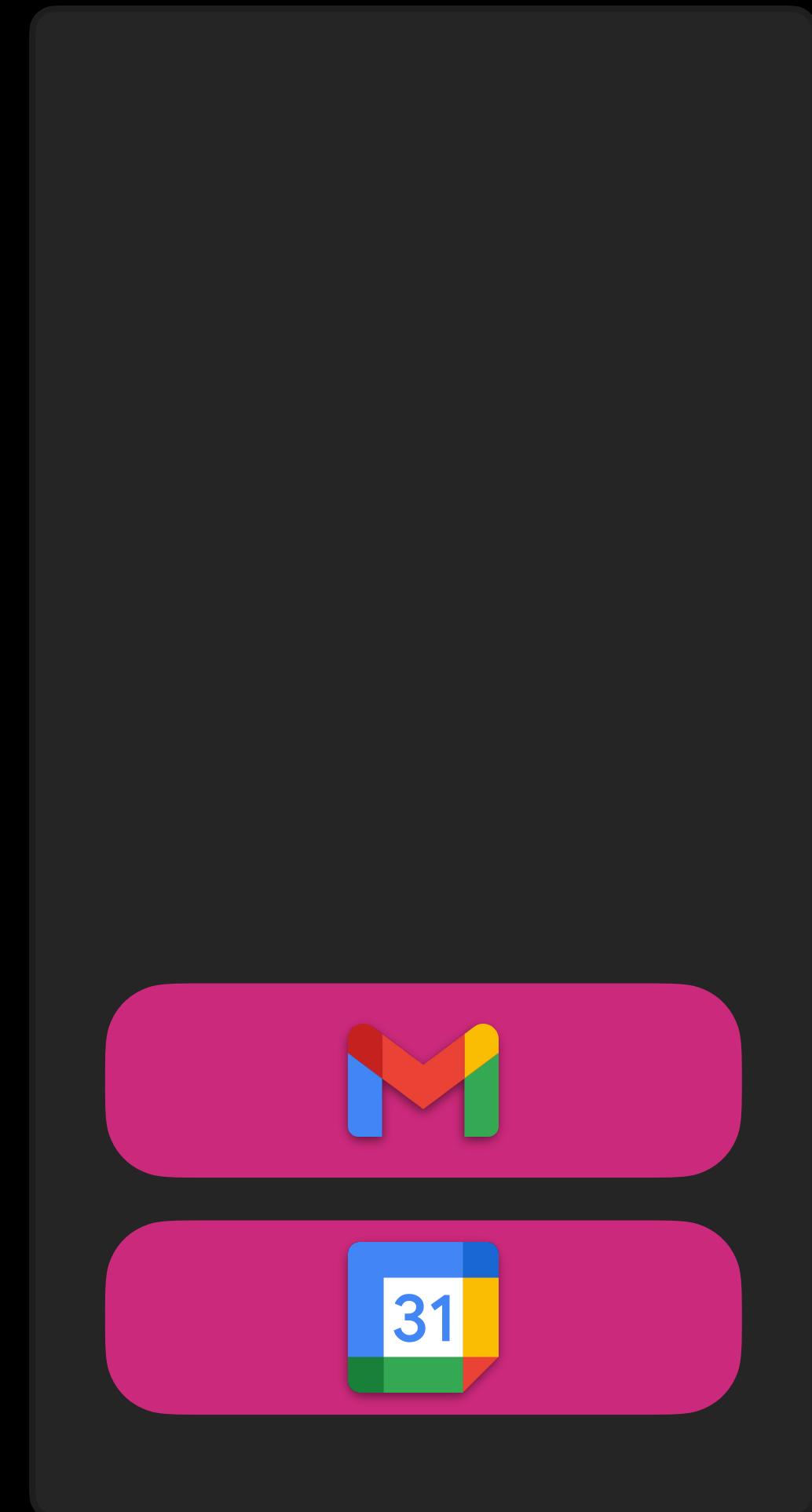
Client

- 1 Richiede una risorsa
- 2 Restituisce la risorsa
- 3 Richiede la risorsa
- 4 Restituisce la risorsa



Resource Server

- 2 Richiede la risorsa
- 3 Restituisce la risorsa



Authorization Code Flow with PKCE



La punta di diamante di OAuth, anche per client pubblici!

OAuth Flows

Authorization

Code Flow w/ PKCE

Client pubblico

Come l'Authorization Code Flow standard, ma senza `client_secret`. Al suo posto, viene utilizzato un codice random.

```
const code_verifier = base64(randomBytes);
const code_challenge = sha256(code_verifier);
```

Request

```
GET https://accounts.google.com/o/oauth2/auth
?scope=gmail.insert gmail.send
&redirect_uri=https://your-app.com/callback
&response_type=code
&client_id=86151612616
&state=wef5w6ed1w6w
&code_challenge=code_challenge
&code_challenge_method=S256
```

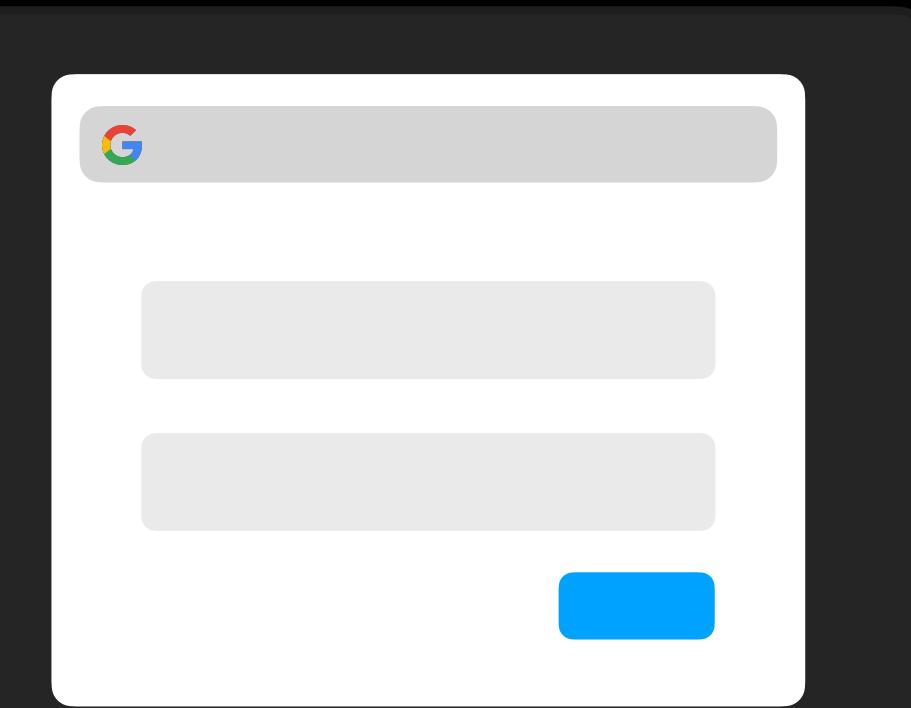
Request

```
POST https://oauth2.googleapis.com/token
code=rg6we16gwedv2w0edvw1d6v
client_id=86151612616
grant_type=authorization_code
code_verifier=code_verifier
```

Client



Authorization Server



- 1 Richiede il **code** /authorize
- 2 Ritorna il **code**
- 3 Richiede l'**Access Token** /token
- 4 Ritorna l'**Access Token**

OAuth Flows

Authorization Code Flow w/ PKCE

- Sostituisce l'Implicit Flow per i Client pubblici (mobile apps, spa...).
- È più sicuro perché non c'è mai un token nell'url. C'è invece il *code*, ma è one-time, quindi innocuo.
- È considerato meno sicuro di un Authorization Code classico, perché i token vengono salvati nel Client...
- ...quindi vanno prese delle precauzioni obbligatorie:
 - Assicurarsi che il Server utilizzi Refresh Token Rotation
 - Assicurarsi che il Server utilizzi Reuse Strategy
- È tecnicamente possibile passare i token a un back-end di appoggio

OAuth Flows

Linee guida per le SPA

- Se l'architettura è semplice (front-end e back-end sullo stesso dominio), la scelta più sicura e più pratica è evitare OAuth e affidarsi ai classici Cookie per mantenere la sessione.
- Le SPA senza back-end devono implementare l'Authorization Code w/ PKCE, e l'Authorization Server deve supportarlo.
- Le SPA devono usare il parametro **state**, generarne uno nuovo per ogni richiesta, e verificare che il valore di ritorno coincida.
- Se proprio la SPA deve ricevere un Refresh Token, l'Authorization Server deve per lo meno supportare Refresh Token Rotation e Reuse Strategy.

OpenID Connect



Ottieni informazioni sul profilo dell'utente, protocollo basato su OAuth 2.0

OAuth 2.0

Protocollo di Autorizzazione

OpenID Connect

Protocollo di Autenticazione (estensione di OAuth 2.0)

Prima di OIDC...

Pseudo-authentication

GET `https://authorizationserver/me`

*Endpoint custom che, dato un Access Token,
ritorna il **profilo** utente... non standard.*

OAuth Flows

Ricordi i flow? Basta aggiungere lo scope **openid**

Request

```
GET https://authorizationserver  
?scope=... openid  
&redirect_uri=...  
&response_type=...  
&client_id=...  
&state=...  
&code_challenge=...  
&code_challenge_method=S256
```

Un AS che supporta OIDC viene chiamato **Identity Provider**

OAuth Flows

Otterrai in aggiunta un **ID Token** in formato JWT (leggibile dal client)

```
{  
  "access_token": "wefwe61fwvs2",  
  "token_type": "Bearer",  
  "expires_in": 3600,  
  "refresh_token": "dstgn541sns61g",  
  "id_token": "JWT"  
}
```

ID Token

Esempio (decodificato)

```
{  
  "iss": "http://my-domain.auth0.com",  
  "sub": "auth0|123456",  
  "aud": "my_client_id",  
  "exp": 1311281970,  
  "iat": 1311280970,  
  "name": "Jane Doe",  
  "given_name": "Jane",  
  "family_name": "Doe",  
  "gender": "female",  
  "birthdate": "0000-10-31",  
  "email": "janedoe@example.com",  
  "picture": "http://example.com/janedoe/me.jpg"  
}
```

OIDC Scopes

Ottieni più informazioni (*claim*) sull'utente con scope aggiuntivi

SCOPES	CLAIMS
email	email, email_verified
phone	phone_number, phone_number_verified
profile	name, family_name, given_name, middle_name, nickname, preferred_username, profilepicture, website, gender, birthdate, zoneinfo, locale, updated_at
address	address

/userinfo



Endpoint standard che, dato un Access Token, ritorna le stesse informazioni dell'ID Token

JWT



JSON Web Tokens

JSON Web Tokens

Cos'è un JWT?

I JSON Web Token ( jot) sono uno standard (RFC 7519) usato per rappresentare dei claim (informazioni, permessi) e poterli inviare in modo sicuro ad applicazioni di terze parti.

Sono la tipologia di token più usata in OAuth 2.0 e OIDC per rappresentare sia gli Access Token che gli ID Token.

Un JWT, una volta creato, non può essere modificato: nel caso un attaccante provasse a manometterlo, il token risulterebbe non valido.

JSON Web Tokens

Cos'è un JWT?

eyJhbGciOiJIUzI1
NiIsInR5cCI6Ikpx
VCJ9.eyJzdWIi0iI
xMjM0NTY30DkwIiw
ibmFtZSI6Ikpvag4
gRG9LIiwiawF0Ijo
xNTE2MjM5MDIyfQ.
SfLKxwRJSMeKKF2Q
T4fwpMeJf36P0k6y
JV_adQssw5c

Header

Informazioni sul tipo di token e algoritmo usato

Payload

Insieme di claim (informazioni sullo specifico token). Può contenere fra le altre cose la data di creazione e di scadenza del token.

Signature

Firma del token, generata dall'algoritmo specificato nell'Header. Comprende un valore segreto.

```
{  
  "alg": "HS256",  
  "typ": "JWT"  
}  
  
{  
  "sub": "1234567890",  
  "name": "John Doe",  
  "iat": 1516239022  
}  
  
HMACSHA256(  
  base64UrlEncode(header) + "." +  
  base64UrlEncode(payload),  
  your-256-bit-secret  
)
```

JSON Web Tokens

Claims

Esistono 3 tipologie di claim: Reserved, Public, Private.

- I Reserved Claim sono specificati dallo standard, ad esempio: `sub`, `iss`, `iat`, `exp`...
- I Public Claim non sono parte dello standard JWT ma sono standardizzati da un'organizzazione chiamata IANA, ad esempio: `given_name`, `family_name`, `preferred_username`...
- I Private Claim sono a libera scelta. Non inserire informazioni sensibili.

JSON Web Tokens

Esempio di Access Token

```
{  
  "iss": "https://foo.com",      // Issuer: chi ha creato il token  
  "sub": "ehw6dvb6sb1",        // Subject: quale utente rappresenta  
  "iat": 1633313441,           // Issued At: data di creazione  
  "exp": 1633319641,           // Expiry: data di scadenza  
  "scope": "openid profile",   // Scope: quali permessi ha l'utente  
  ...                          // Altro...  
}
```

JSON Web Tokens

Esempio di ID Token

```
{  
  "iss": "...", "sub": "...", "iat": "...", "exp": "...",  
  "nickname": "MicheleStieven",  
  "name": "Michele Stieven",  
  "email": "foo@bar.com",  
  "picture": "https://foo.com/michele.jpg",  
  "updated_at": "2021-10-10T11:00:00.000Z"  
  ...  
}
```

Verifica di un JWT

Un JWT può essere validato facilmente avendo a disposizione il suo **segreto**. Basta rigenerare la firma e verificare che il token coincida. Se header o payload fossero modificati, cambierebbe anche la firma.

```
const token =  
  base64urlEncode(header) + '.' +  
  base64urlEncode(payload) + '.' +  
  base64urlEncode(signature)
```

A fini di esempio puoi
verificare un token sul
sito [JWT.io](#)



Firma asimmetrica

Un JWT può essere firmato in modo simmetrico (con un segreto) o asimmetrico (chiave privata e chiave pubblica). La chiave privata rimane segreta, quella pubblica può essere usata per la verifica.

Simmetrici

- HMACSHA256, HMACSHA384, HMACSHA512 ...

Asimmetrici

- RSASHA256, RSASHA384, RSASHA512, ECDSASHA256 ...

JSON Web Tokens

.well-known

Un Authorization Server OIDC mette a disposizione un endpoint `/well-known/openid-configuration` contenente molte informazioni sul server (ad esempio, gli endpoint di autenticazione, del token, ecc...). Viene molto usato, specialmente dalle librerie di autenticazione per evitare allo sviluppatore di inserire tutto a manualmente.

A questo indirizzo si può trovare una voce `JWKS_URI`, che fornisce un URL dove trovare il `JWK` (`JSON Web Key`), ovvero un oggetto contenente le chiavi pubbliche per verificare i token.

JSON Web Tokens

Recap

- Un JWT non può essere modificato
- Può essere usato sia per gli Access Token che gli ID Token (OpenID Connect)
- Può contenere informazioni a piacimento, ma non devono essere sensibili
- Può essere scambiato fra applicazioni su domini completamente differenti
- Contiene dentro di sé la data di scadenza: nel caso di revoca del token (dipendente licenziato, cambio di ruolo, logout...) la soluzione non è semplice, serve una blacklist sul server
- BONUS: è anche possibile criptare un JWT tramite JWE (JSON Web Encryption), ma non tutti lo fanno perché non è detto serva

Token senza OAuth



Soluzione custom... ne vale la pena?

Critiche frequenti ai Cookie

Spoiler: quasi tutte infondate

- Meno sicuri (CSRF)
- Non scalano
- Non funzionano su mobile
- Difficili da usare
- Non funzionano fra domini differenti

Meno sicuri?

Sbagliato.

- ✓ Le vulnerabilità di tipo CSRF sono facilmente risolvibili con l'attributo SameSite e, optionalmente, un CSRF Token.

Non scalano?

Spoiler: qualsiasi big (Google, Facebook...) ne fa uso.

- Più processi su un solo server...
 -  Redis
- Più server...
 -  Redis (su server dedicato)
- Più server, più cluster...
 -  Sticky Sessions

Non funzionano su mobile?

Sbagliato.

- ✓ Ogni browser mobile supporta i Cookie.
- ✓ App native: ogni framework mobile supporta i Cookie.

Difficili da usare?

Sbagliato.

- ✓ Per molti sviluppatori sembrano difficili solo perché non li hanno mai usati: in realtà, il meccanismo è semplicissimo.
- ✓ Ideali per approcci SSR (Server-side rendering), dove un token non verrebbe inviato al server alla prima richiesta.

Non funzionano fra domini differenti?

Vero.

- ✓ ...Ma è proprio necessario che il tuo server sia su un dominio differente?
- ✓ In molti casi, si può valutare di usare un Proxy.

Complessità dei token

- Ogni chiamata al server pesa di più (un Cookie è al massimo 4kb)
 - Servono chiamate aggiuntive per ogni refresh, per ogni utente...
 - ...e il front-end deve implementarlo, o l'utente verrà disconnesso spesso
- Il server deve sia generare che validare i token
- Non sono realmente stateless: serve una blacklist di Refresh Token invalidi
 - ...e una per gli Access Token, se vuoi che una modifica abbia effetto immediato
- Serve maggior protezione, dovrai implementare:
 - Refresh Token Rotation
 - Reuse Detection

Piuttosto, usa Auth0 / Okta

Gestiscono tutto questo e molto di più (Social Login, User management...)



Auth0 non è OAuth

Auth0 è una **compagnia** che si occupa di autenticazione / autorizzazione al posto tuo. Utilizza OAuth 2.0 e OIDC.

SPA Authentication, con OAuth e OIDC

La SPA è senza **backend**?

Browser gets tokens

Authorization Code w/ PKCE
prende il posto di Implicit Flow

Per mantenere l'utente loggato, avete due strade: se il **front-end** e l'**Authorization Server** sono sullo stesso dominio (o sottodomini), potete utilizzare il **Silent Renew**. Altrimenti, impostare lo scope "offline_access" per ricevere un **Refresh Token**. In questo caso, molta attenzione! **Refresh Token Rotation** e **Reuse Strategy** vanno abilitate per sicurezza sull'Authorization Server. Non tutti li supportano!

Vuoi solo il **profilo** utente?

Server gets ID Token

Implicit Flow with Form Post

NON è la stessa cosa dell'Implicit Flow. In questo flow, viene fatto un redirect al **back-end** mettendo l'**ID Token** in POST. Il back-end lo riceve, fa un redirect alla SPA e **setta un Cookie** per tenerla autenticata (inserendoci l'ID Token o un identificativo, tenendosi l'ID Token in memoria). Nessun refresh in questo flow, nessun Access Token. In alternativa, usare l'**Authorization Code Flow** e non considerare l'Access Token.

COOKIE

Vuoi accedere alle **API** del provider, oltre al profilo?

Server gets tokens

Authorization Code Flow
PKCE opzionale

Per mantenere l'utente loggato, basta un **Cookie** fra FE e BE (con un identificativo, e il BE tiene i token in memoria). È il BE che si occupa di aggiornare l'Access Token quando scade, perché ha un **Refresh Token** per ogni utente. Il FE non chiama direttamente le API, ma chiama il BE che a sua volta chiama le API. Questa architettura viene chiamata **Back-end for Front-end**, potete eventualmente impostare un Proxy sul BE per comodità. È la strada **più sicura**.

COOKIE

ALTERNATIVA

Browser gets tokens

Authorization Code w/ PKCE
prende il posto di Implicit Flow

Il BE non sa dell'autenticazione che avviene interamente sul FE, bisogna avvisarlo: al **success** del login, **inviare l'ID Token al BE** (con CSRF Token!). Il BE deve **verificare** che il Token soddisfi i requisiti dell'Authorization Server. Se è ok, si setta un **Cookie** con il FE (se è la prima volta, crea un nuovo utente nel database). Il **FE ha Access Token e Refresh Token** quindi può fare le chiamate API per conto suo. Al **logout** dal provider, togliere anche questo Cookie! Questa soluzione è più comoda per il BE, ma **meno sicura** perché i token sono nel browser.

COOKIE

Queste considerazioni continuano a valere!

FAQ

- Se tutto avviene nello stesso dominio, un'autenticazione via Cookie è **sempre da preferire**: non c'è bisogno di alcun token!
- Quando si parla di Cookie di autenticazione, il Cookie deve obbligatoriamente avere i flag **HttpOnly** e **Secure**: non dev'essere visibile da JavaScript e deve funzionare solo in HTTPS. Altri flag (come SameSite, Domain, Path...) possono essere settati per maggiore sicurezza. L'unico Cookie che non deve essere HttpOnly è quello contenente il **CSRF Token**, che deve essere visibile da JavaScript e inviato ad ogni richiesta.
- È possibile utilizzare il **Code Flow w/ PKCE** se l'Authorization Server non usa **Refresh Token Rotation** e **Reuse Strategy**? È possibile, ma **altamente sconsigliato**. Se possibile, utilizzare il **Silent Renew** ed evitare completamente i Refresh Token, e se l'Authorization Server è in un altro dominio valutare di agire sui DNS per impostare un **dominio personalizzato** (ad esempio, **Auth0** e **Okta** lo permettono). I browser stanno facendo passi avanti in materia di sicurezza e il Silent Renew non funzionerà più fra domini differenti, vedi “Safari ITP”.
- È possibile, alla fine di un **Authorization Code Flow**, ritornare i token al front-end al posto di settare un Cookie? È possibile, ma il processo **diventa meno sicuro**. Dare un Refresh Token al browser rappresenta sempre un rischio (anche minimo) di sicurezza, e diamo per scontato che **RTR** e **Reuse Strategy** vengano utilizzate. Più o meno è quello che fa **Firebase** nel suo **SDK JavaScript**: utilizza il **Code Flow** per andare sul back-end del nostro progetto (`__/auth/handler`) ma poi restituisce i token al browser. È un approccio utile per rendere il front-end autonomo e rende il setup più veloce perché funziona su domini differenti.
- Nel momento in cui scalo la mia applicazione e introduco un **Load Balancer**, perdo la possibilità di usare i Cookie? No, puoi usare le **Sticky Session** o, ancora meglio, un server (o cluster) ad hoc con **Redis**, condiviso fra tutte le istanze del tuo back-end. Qualsiasi big usa i Cookie: Google, Facebook...
- A cosa mi servono servizi a pagamento come **Auth0** e **Okta** se posso autenticarmi gratuitamente con Google/Facebook/ecc? Questi servizi vi forniscono un **Authorization Server** senza il bisogno di mettere in piedi voi stessi un server OAuth2.0/OIDC (molto laborioso), il Social Login è solo la punta dell'iceberg.
- Utilizzare i token, nella realtà, **non rende il server stateless**: va mantenuto uno stato (black-list) per revocare i token e per le altre tecniche di sicurezza.