OAuth2 in the wild

OAuth in web and mobile applications

Eyad Issa

Sicurezza dell’Informazione M
Corso di Laurea Magistrale in Ingegneria Informatica

15/07/2025

Contents

1. The OAuth Standard

1.1 The OAuth Standard

RFC 6749 *(The OAuth 2.0 Authorization Framework)*[1] was published in 2012 .

Back then, most web apps followed a classic **server-rendered model**. Page loads were full refreshes, and all navigation went through the backend (PHP, Rails, Spring, …).

[1] RFC 6749

1.1 The OAuth Standard

Before OAuth, apps needed users to hand over their credentials to act on their behalf.

This was a **huge security risk**: credentials had to be **stored in plain text** because they needed to be used, not just verified.

With OAuth, users finally gained control:

- Grant limited access to applications;
- Keep credentials secret.

1.2 OAuth 2 Flows

OAuth 2 introduces multiple flows (also called “grants”), which define how different parties interact according to the standard:

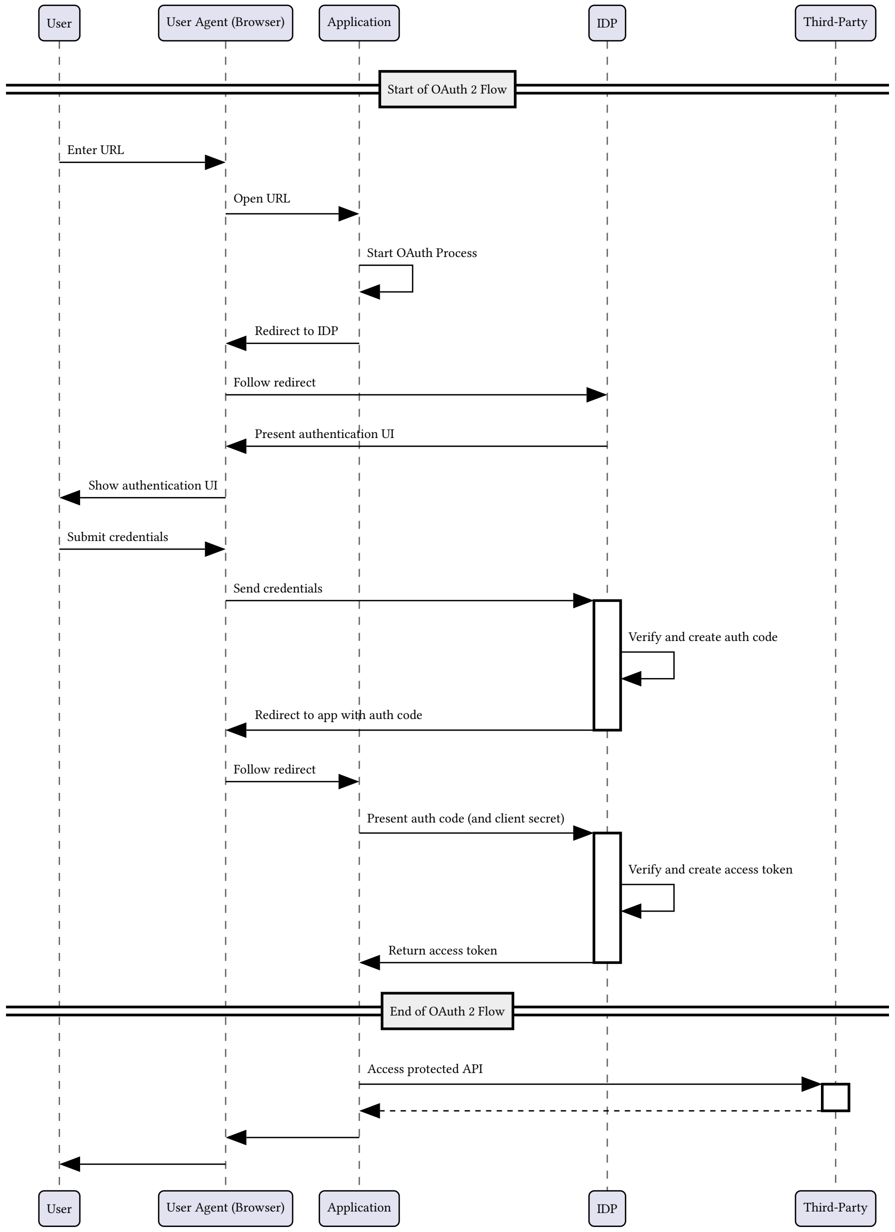
- Authorization Code Flow
- PKCE
- Client Credentials Flow
- Device Code Flow
- Refresh Token Flow
- **Legacy:** Implicit Flow
- **Legacy:** Password Grant Flow

1.3 Authorization Code Flow

The **Authorization Code Flow** is the classic OAuth 2.0 pattern.

In this flow, **a user authorizes a client to act on their behalf** with a third party.

An **Identity Provider** handles the majority of the process, authenticating the user, issuing **authorization codes** and then **access tokens**.


User
User Agent (Browser)
Application
IDP
Third-Party
Start of OAuth 2 Flow
Enter URL
Open URL
Start OAuth Process
Redirect to IDP
Follow redirect
Present authentication UI
Show authentication UI
Submit credentials
Send credentials
Verify and create auth code
Redirect to app with auth code
Follow redirect
Present auth code (and client secret)
Verify and create access token
Return access token
End of OAuth 2 Flow
Access protected API
User
User Agent (Browser)
Application
IDP
Third-Party

1.4 It is not enough?

This process comes with a few challenges:

- The application must be a ***Confidential Client***:
  - It needs a secure way to store a **client secret**.

- The flow only covers the main goal of OAuth 2.0, which is to **authorize a client**.
  - It doesn't handle the session between the user agent and the web application backend.
  - A separate protocol is required to manage that part.

2. OAuth2 on the (modern) web

2.1 The advent of SPAs

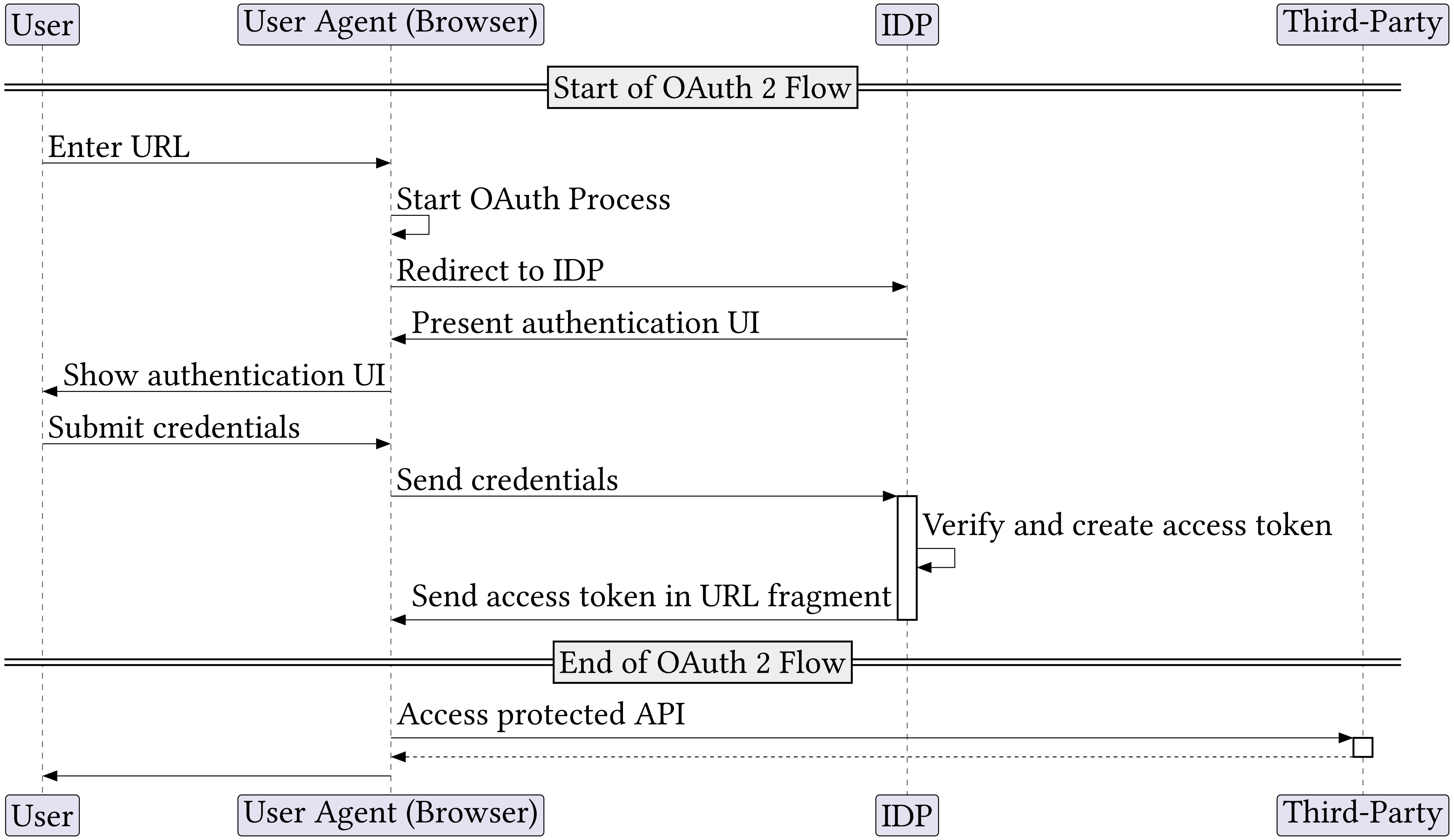
Single Page Applications (SPAs) have revolutionized the way web applications are built.

- SPAs dynamically rewrite the DOM rather than loading entire new pages from the server.
- State transitions are entirely handled on the client side
- The backend can be streamlined to focus primarily on CRUD (Create, Read, Update, Delete) operations, complemented by robust authentication and authorization mechanisms.

2.2 Implicit Flow

A novel workflow has been designed to facilitate the adoption of OAuth in Single Page Applications (SPAs), particularly in scenarios where a backend is not present:

The **Implicit Flow**


User
User Agent (Browser)
IDP
Third-Party
Start of OAuth 2 Flow
Enter URL
Start OAuth Process
Redirect to IDP
Present authentication UI
Show authentication UI
Submit credentials
Send credentials
Verify and create access token
Send access token in URL fragment
End of OAuth 2 Flow
Access protected API
User
User Agent (Browser)
IDP
Third-Party

2.4 Implicit vulnerabilites

The Implicit Flow was revealed to be weak. An attacker could intercept the redirect and gain access to the access code.

Commonly this was a big issue on mobile applications, where an attacker could just register a URL handler.

```
http://myapp.com/#/redirect?access_token=<ACCESS_TOKEN>
```

2.5 The Advent of PKCE

Proof Key of Code Exchange was created to **secure public clients**, as an addition to the classic Authorization Code Flow.

- No need for a Client Secret
- **Secure storage still needed**

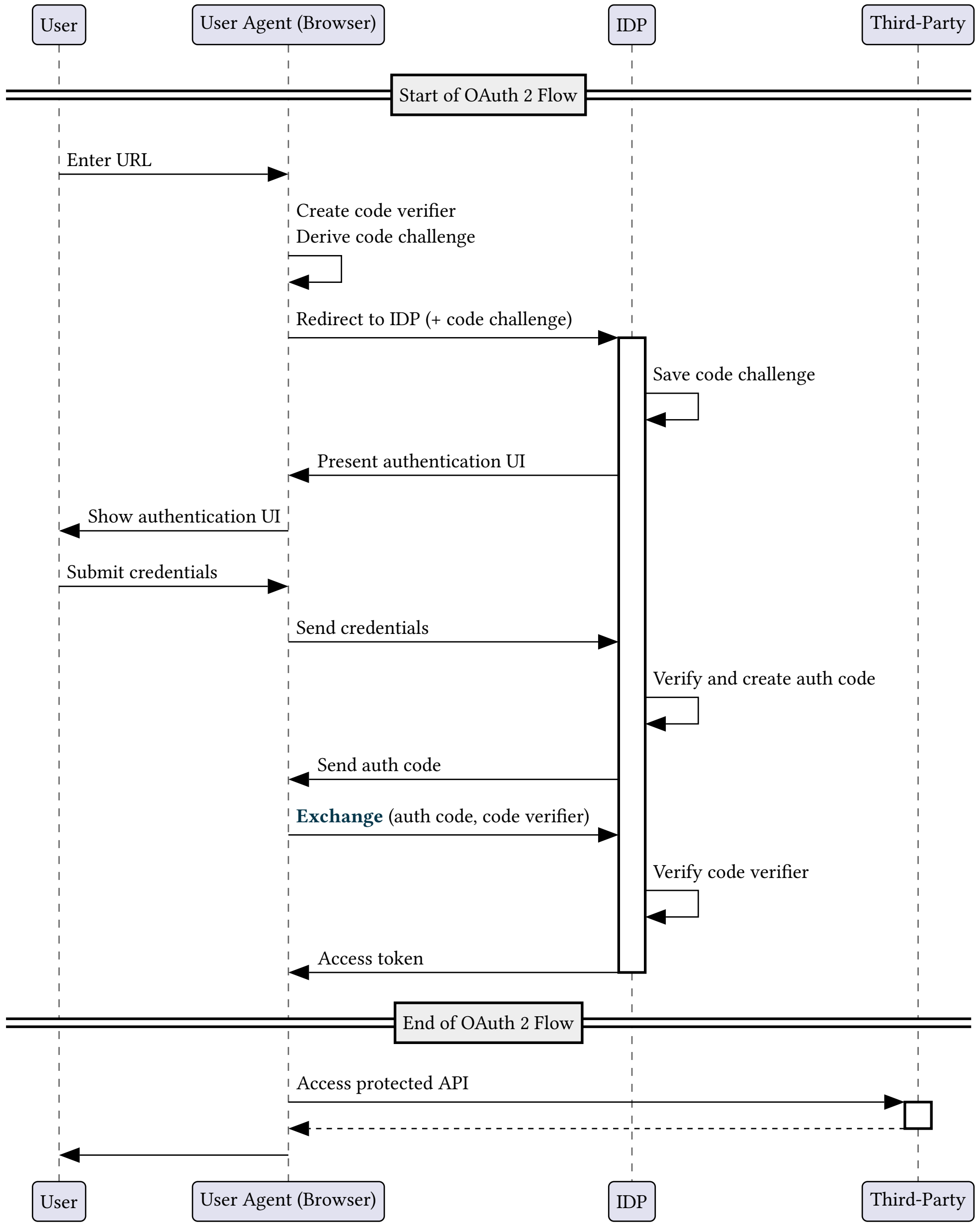
2.5 The Advent of PKCE

Relies on the irreversibility of hashes for its operation:

code verifier A cryptographically random string that is used to correlate the authorization request to the token request.

code challenge A challenge derived from the code verifier that is sent in the authorization request, to be verified against later.

code challenge method A method that was used to derive code challenge.


User
User Agent (Browser)
IDP
Third-Party
Start of OAuth 2 Flow
Enter URL
Create code verifier
Derive code challenge
Redirect to IDP (+ code challenge)
Save code challenge
Present authentication UI
Show authentication UI
Submit credentials
Send credentials
Verify and create auth code
Send auth code
Exchange (auth code, code verifier)
Verify code verifier
Access token
End of OAuth 2 Flow
Access protected API
User
User Agent (Browser)
IDP
Third-Party

2.6 The lack of a safe storage

The absence of secure storage on web platforms arises from the potential for an attacker to inject arbitrary JavaScript code into the DOM, using attacks such as Cross-Site Scripting (XSS), which are challenging to completely mitigate.

- **Cookies:** Unsafe, can be accessed by `document.cookies`. Only `HttpOnly` cookies are safe, but the need to adopt a Backend for Frontend pattern arises;
- **Local Storage:** Totally unsafe, can be accessed by JS.

2.7 Various models

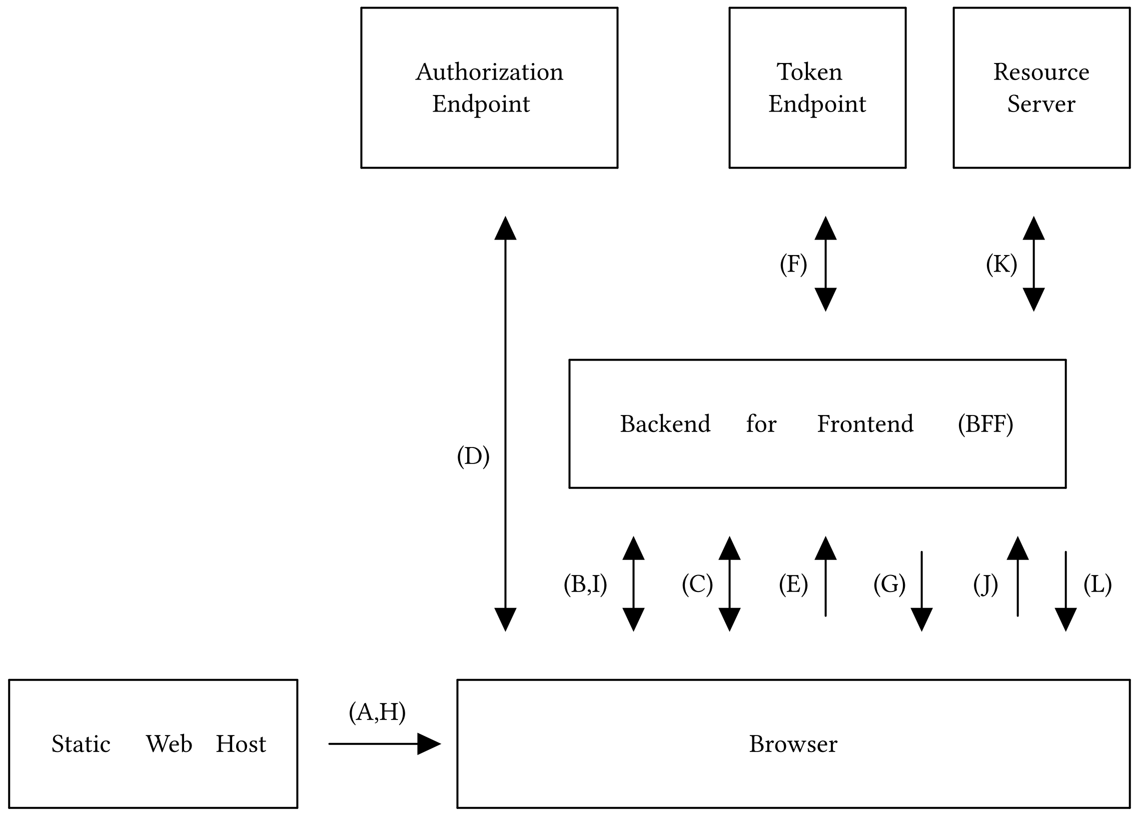
The latest *OAuth 2.0 for Browser-Based Applications*[1] internet draft details the threats, attack consequences, security considerations and best practices that must be taken into account when developing browser-based applications that use OAuth 2.0.

2.7.1 Recommended Application Architecture Patterns

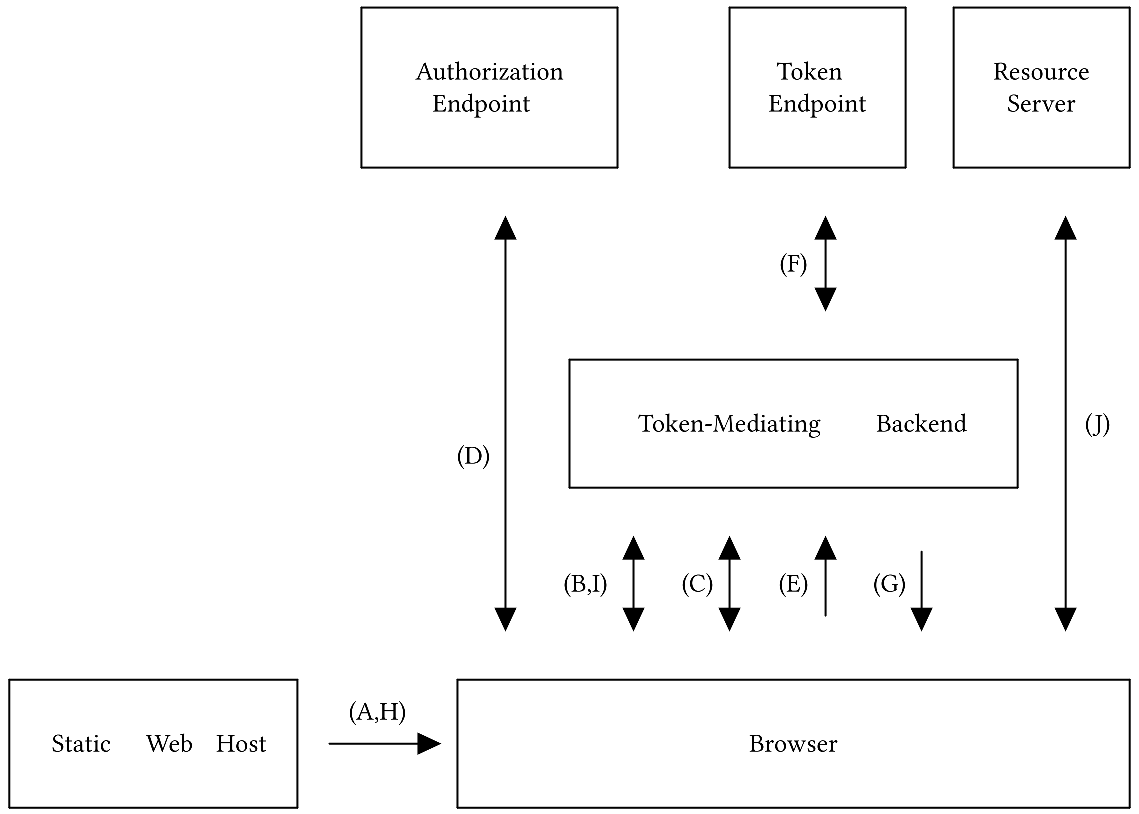
- Backend For Frontend (BFF)
- Token-Mediating Backend
- Browser-based OAuth 2.0 client

[1] https://datatracker.ietf.org/doc/html/draft-ietf-oauth-browser-based-apps

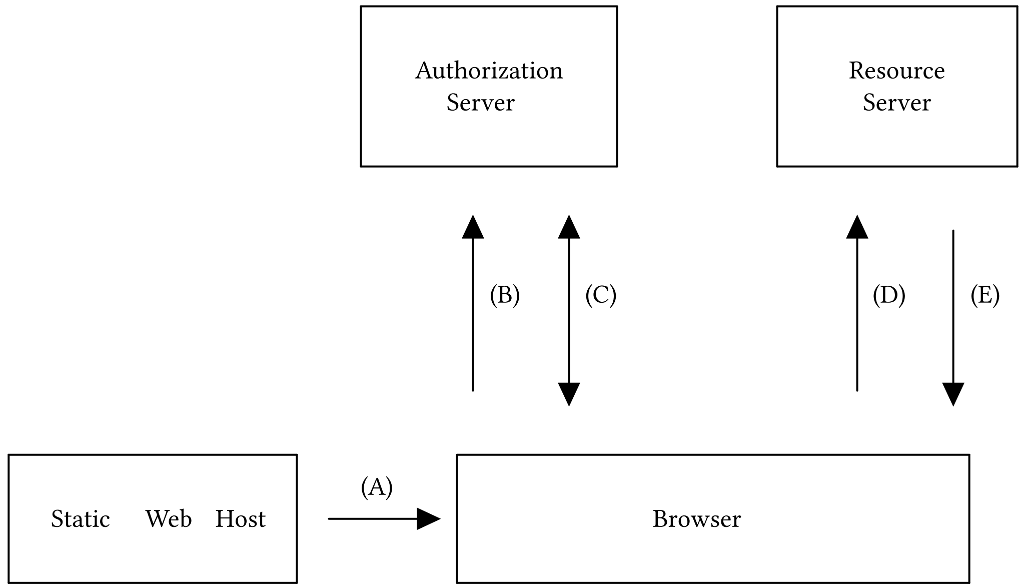
2.8 Backend For Frontend (BFF)



2.9 Token Mediating Backend



2.10 Browser-based OAuth 2.0 client



2.11 Web workers FTW ?

Section 6.3.4.2.1. of the latest recommendations [1] says:

> When handling tokens directly, the application can choose different storage mechanisms to store access tokens and refresh tokens.
>
> Universally accessible storage areas, such as Local Storage, are easier to access from malicious JavaScript than more isolated storage areas, such as a Web Worker.

3. Mobile based OAuth2

3.1 Intercepting web requests

A mobile application aiming to function as an OAuth2 client must mimic a web application by intercepting the redirect callback.

There are two primary methods to achieve this:

- Using an **Embedded WebView**
- **Registering specific URLs** for interception through the operating system

3.2 Embedded WebView

Despite being discouraged by the official standard[1], using embedded WebViews to implement an OAuth2 flow is still common among mobile applications.

[1] https://www.oauth.com/oauth2-servers/oauth-native-apps/use-system-browser/

3.2 Embedded WebView

There are multiple issues with this, in particular:

- the client app can potentially **eavesdrop on the user entering their credentials** when signing in, or even **present a false authorization page**.
- **the embedded web view doesn't share cookies** with the system's native browser, so users have a worse experience because they need to enter their credentials each time as well.

3.3 Registering URLs

Some platforms, (Android, and iOS as of iOS 9), allow the app to override specific URL patterns to launch the native application instead of a web browser.

For example, an application could register

```
https://app.example.com/auth
```

and whenever the web browser attempts to redirect to that URL, the operating system launches the native app instead.

3.3 Registering URLs

If the operating system does some level of validation that the developer had control over this web URL, then this allows the identity of the native application to be guaranteed by the operating system[1].

```
https://example.com/.well-known/assetlinks.json
```

[1] https://developer.android.com/training/app-links/verify-android-applinks

3.4 Registering custom protocols

Most mobile and desktop operating systems allow apps to **register a custom URL scheme** that will launch the app when a URL with that scheme is visited from the system browser.

The redirect URL will be a URL with the app's custom scheme.

```
myapp://callback#token=....
```

Unfortunately, the OS has no way of ensuring the developer is the righful owner of the scheme, so a malicious app could present itself as a choice for the protocol.

4. A case study: MyUnibo

4.1 The mobile (kind of) application

Tech stack:

- Powered by **React Native**
- **Hermes:** Custom JS runtime (and bytecode)

4.1.1 Reversing

1. Download and unzip the APK.
2. `index.android.bundle` contains the hermes bytecode.
3. hermes-dec to reverse the bundle to JS

4.2 Embedding the views

Uses React Native Webviews and `onNavigationStateChange` to extract the callback.

```
https://myunibo.unibo.it/signin?...
```

```
❯ dig myunibo.unibo.it
NXDOMAIN
```

Uses Authorization Grant with PKCE

$\longrightarrow$ Client ID is public!

5. Demo time

6. Securing OAuth2: Proof of Possession

6.1 Mutual TLS

https://tools.ietf.org/html/rfc8705

MTLS is a form of client authentication and an extension of OAuth 2.0 that provides a mechanism of binding access tokens to a client certificate.

6.2 DPoP

datatracker.ietf.org/doc/html/rfc9449

DPoP, or Demonstrating Proof of Possession, is an extension that describes a technique to cryptographically bind access tokens to a particular client when they are issued.

Bibliography

[1] A. Parecki, P. D. Ryck, and D. Waite, “OAuth 2.0 for Browser-Based Applications,” Internet Engineering Task Force, Jul. 2025. [Online]. Available: https://datatracker.ietf.org/doc/draft-ietf-oauth-browser-based-apps/25/