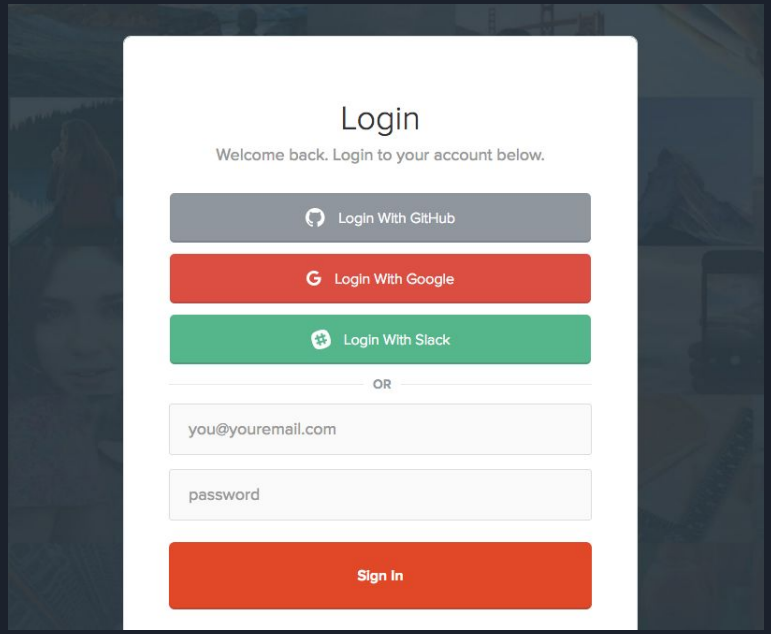
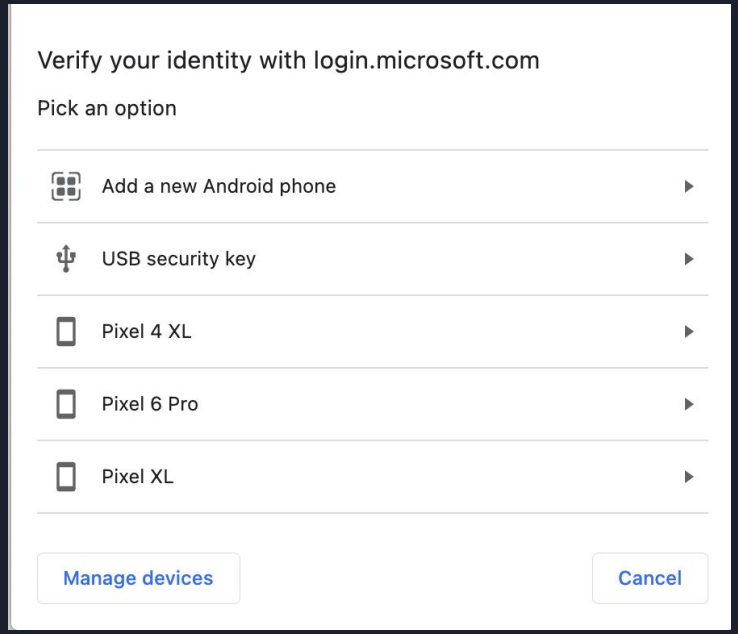
Interactive Authentication of Non-Interactive HTTP Requests

Ben Schwartz, HTTPAPI and OAUTH @ 115

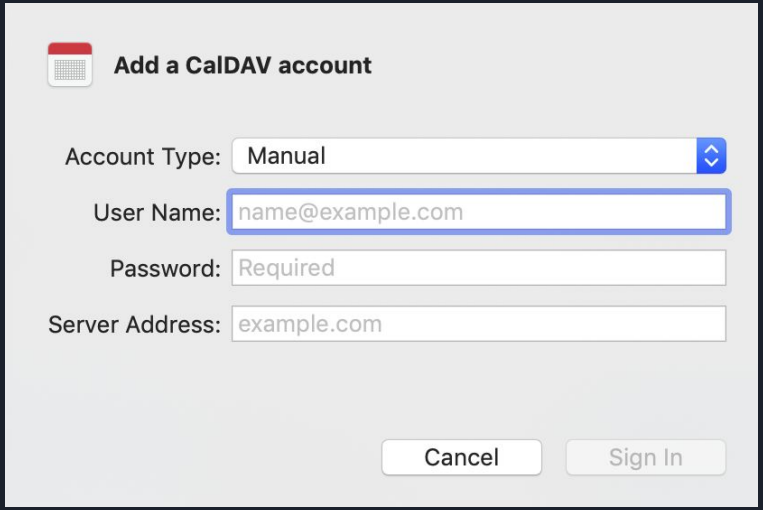
Popup Authentication

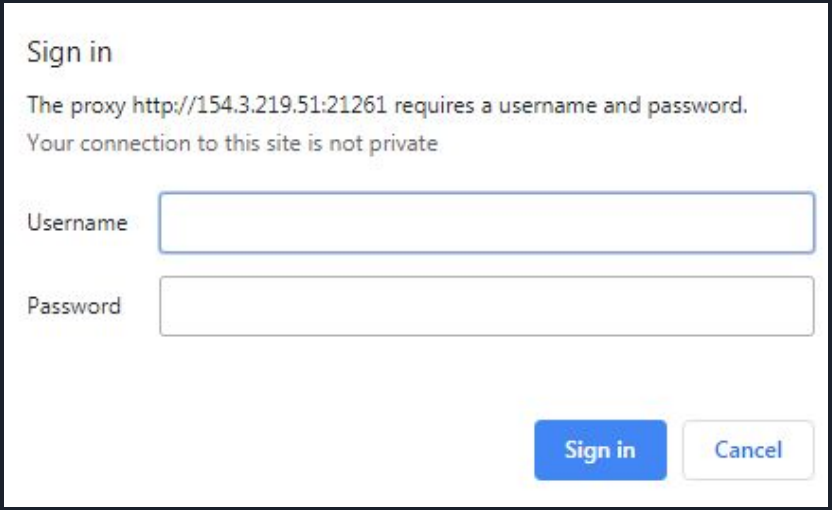
This is what login looks like on the web today...





… and this is how it looks for the rest of HTTP





Non-web HTTP login is stuck in 1996

What about OAuth?

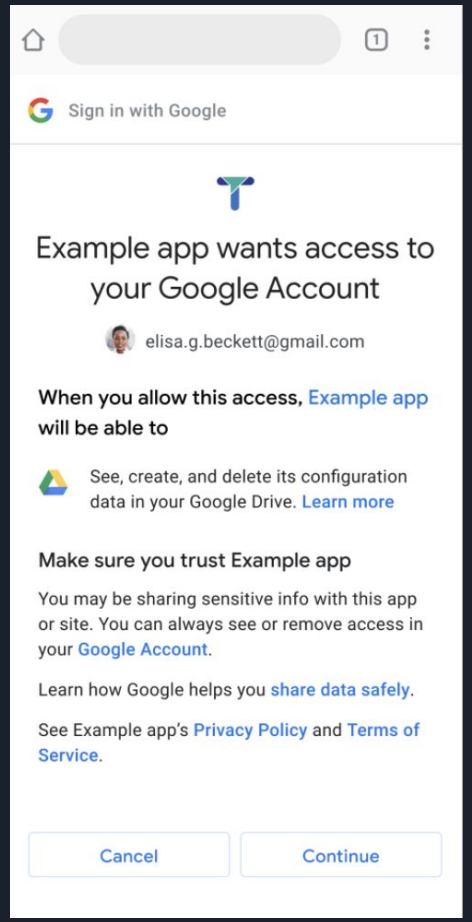
OAuth (currently) enables clients to speak proprietary protocols (over HTTP) to specific origins that are known in advance.

This protocol is for clients that want to speak standardized protocols (over HTTP) to any compatible origin.

Your Example service has requested interactive authentication.

OPEN BROWSER

CHANGE EXAMPLE PROVIDER



Interactive authentication complete

OK

HTTP Exchange 1: The trigger

```
OPTIONS /home/bemasc/calendars HTTP/1.1
Host: cal.example.com

HTTP/1.1 401 Unauthorized
WWW-Authenticate: interactive location=/login
WWW-Authenticate: ...
```

Hey, do you support CalDAV?

Who are you? Open a browser.

HTTP Exchange 2: The login screen

```
GET /login HTTP/1.1
Host: cal.example.com
Accept: text/html,...
Accept-Language: en-US,...
Sec-Fetch-Dest: document
Sec-Fetch-Mode: navigate
Sec-Fetch-Site: none
Sec-Fetch-User: ?1
...

HTTP/1.1 401 Unauthorized
Content-Type: text/html
...
```

Hi cal.example.com, this is the web browser.

The user has opened https://cal.example.com/login in a new tab.

Here are the login instructions.

HTTP Exchange 3: The success signal

```
GET /login HTTP/1.1
Host: cal.example.com
Accept: text/html,...
Accept-Language: en-US,...
Sec-Fetch-Dest: document
Sec-Fetch-Mode: navigate
Sec-Fetch-Site: same-origin
Sec-Fetch-User: ?1
Cookie: login=6bb0e2c8-874e-44c8-b8e0-25e12f339b46
...

HTTP/1.1 200 OK
Content-Type: text/html
...
```

The user followed a link to https://cal.example.com/login, and I already have a cookie for this request.

OK, you can close now.

HTTP Exchange 4: The access

```
OPTIONS /home/bemasc/calendars HTTP/1.1
Host: cal.example.com
Cookie: login=6bb0e2c8-874e-44c8-b8e0-25e12f339b46

HTTP/1.1 200 OK
Allow: OPTIONS, GET, HEAD, POST, PUT, DELETE, TRACE, COPY, MOVE
Allow: PROPFIND, PROPPATCH, LOCK, UNLOCK, REPORT, ACL
DAV: 1, 2, access-control, calendar-access
...
```

Hey, do you support CalDAV?
I have a cookie.

Oh hi again.
Yes, I do support CalDAV!

Specified procedure

1. New auth-scheme “interactive” with a “location=” parameter that provides the **authentication path**.
2. The client reacts by opening this path in a browser “popup”.
3. The client interacts, navigates, types passwords, accesses second factors, etc.
4. If the **authentication path** ever loads successfully, the client stores the request headers and closes the popup.
5. The client copies any stored **`Cookie`** or **`Authorization`** headers into its future requests for this origin.

Interesting corners of this spec

- Both Cookie and Authorization headers are supported.
  - “Authorization” is more natural, but only “Cookie” can be used without Javascript.
- Proxy clients convert Authorization into Proxy-Authorization.
  - …but Cookie headers are just dropped
  - Should we define a way to send cookies to a proxy?
- The spec mandates a URL bar (to avoid phishing) and interstitial dialogs before the browser opens and after it closes (to avoid clickjacking).
  - Is there a better way?
- “interactive” can be used alongside “basic” or “digest” for compatibility
  - Browsers are required to ignore “WWW-Authenticate: interactive”
- No way to declare success without closing the browser…

Closing thoughts

- Brand new draft!
- Brings all the goodness of modern web login to the rest of HTTP
- Needs more HTTP and OAuth expert input
  - How should Set-Cookie parameters work?
  - Should we define a way to send cookies to HTTP proxies?
  - Is there a way to share more concepts with OAuth?
- Seeking adoption in HTTPAPI/HTTPBIS/OAUTH/???
- Mentioned in draft-schwartz-masque-access-descriptions as a good way to authenticate to proxies.

Step 1: The Trigger

New Draft

```
OPTIONS /home/bemasc/calendars HTTP/1.1
Host: cal.example.com

HTTP/1.1 401 Unauthorized
WWW-Authenticate: Bearer location="https://authorization-server.com/" scope="read"
```

The specifics of this header are TBD, the important part is it has the full URL of the authorization server. (Should probably follow HTTP Structured Headers tho.)

Note: The authorization server URL could be under the control of the resource server or a completely unrelated server depending on how you want to deploy it.

Step 2: Client Discovers AS Metadata

RFC8414

```
GET https://authorization-server.com/.well-known/oauth-authorization-server HTTP/1.1

HTTP/1.1 200 Ok
Content-Type: application/json

{
  "issuer": "https://authorization-server.com/",
  "authorization_endpoint": "https://authorization-server.com/authorize",
  "token_endpoint": "https://authorization-server.com/oauth/token",
  "registration_endpoint": "https://authorization-server.com/oauth/clients",
  "response_types_supported": "code",
  ...
}
```

Where to open the browser to

Where to get the tokens from

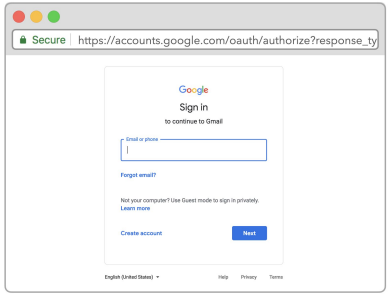
Step 3: Initiate OAuth Flow

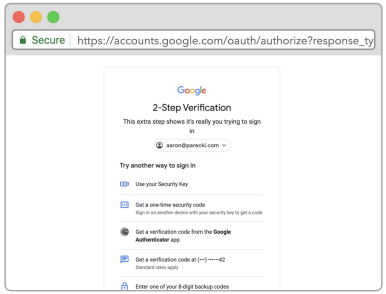
RFC6749

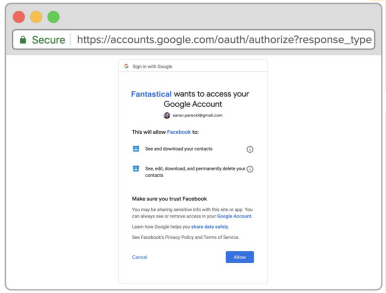
Client launches a browser to initiate the OAuth flow…

```
https://authorization-server.com/authorize?client_id=***&redirect_uri=***&scope=read
   &code_challenge=XXXX&code_challenge_method=S256&state=XXX
```

Normal OAuth flow proceeds, enabling strong MFA and passwordless, as well as SSO







Note: The client_id could be:

- Pre-registered out of band
- Registered dynamically via RFC7591
- Provided as a URI according to a new specification

Note: The redirect_uri could be

- Custom URL scheme
- localhost:port
- "out-of-band"

Step 4: OAuth flow is complete

RFC6749

OAuth flow completes, authorization server redirects to redirect_uri with authorization code, client exchanges code for an access token

```
POST /oauth/token HTTP/1.1
Host: authorization-server.com
Content-type: application/x-www-form-urlencoded

grant_type=authorization_code
&client_id=***
&code_verifier=XXXX

HTTP/1.1 200 OK
Content-type: application/json

{
  "token_type": "Bearer",
  "expires_in": 86400,
  "access_token": "XXXXXXXX",
  "refresh_token": "YYYYYYYYY",
  "scope": "read"
}
```

Note: Refresh token is up to the discretion of the AS, but can be used to get a new token when the current one expires if the AS doesn't need the user to re-authenticate themselves.

Step 5: Resource request

RFC6750

Client uses access token to fetch data

```
GET /home/bemasc/calendars HTTP/1.1
Host: cal.example.com
Authorization: Bearer XXXXXXXXX

CALENDAR DATA RESPONSE
...
```

Note: There are opportunities here to also leverage the new step-up OAuth draft as well, if the RS wants the user to come back with a new or different access token