Workgroup: Web Authorization Protocol
Internet-Draft:
draft-bertocci-oauth2-tmi-bff-01
Published: 25 April 2021
Intended Status: Standards Track
Expires: 27 October 2021
Authors: V. Bertocci B. Campbell
auth0.com Ping Identity

Token Mediating and session Information Backend For Frontend

Abstract

This document describes how a JavaScript frontend can delegate access token acquisition to a backend component. In so doing, the frontend can access resource servers directly without taking on the burden of communicating with the authorization server, persisting tokens, and performing complex operations within the user agent that would require configuration, error management and reliance on authorization server capabilities (such as refresh token rotation) that aren't widely available today.

Status of This Memo

This Internet-Draft is submitted in full conformance with the provisions of BCP 78 and BCP 79.

Internet-Drafts are working documents of the Internet Engineering Task Force (IETF). Note that other groups may also distribute working documents as Internet-Drafts. The list of current Internet-Drafts is at https://datatracker.ietf.org/drafts/current/.

Internet-Drafts are draft documents valid for a maximum of six months and may be updated, replaced, or obsoleted by other documents at any time. It is inappropriate to use Internet-Drafts as reference material or to cite them other than as "work in progress."

This Internet-Draft will expire on 27 October 2021.

Table of Contents

1. Introduction

A large portion of today's development stacks, practices and tools for the web target the user agent itself as execution environment, leveraging local resources to offer a rich, responsive user experience that rivals native applications.

An important aspect of apps running in the user agent is their reliance on HTTP APIs, served from the app's own backend component or from third party providers on disparate domains. Whenever those API are secured according to the OAuth2 Bearer Token Usage

[RFC6750], the user agent app needs to obtain suitable access tokens: however, the task of implementing an OAuth2 [RFC6749] client executing in a user agent is complicated by security challenges and restrictions inherent in the browser platform. The original OAuth2 [RFC6749] provided guidance dedicated to user agent apps in section 4.2, via the implicit grant. The approach proved to suffer from too many challenges, however, leading subsequent documents (such as the OAuth2 security BCP [I-D.ietf-oauth-security-topics] and OAuth 2.1 [I-D.ietf-oauth-v2-1]) to recommend a more secure approach based on the authorization code grant with PKCE [RFC7636], and relying on additional security measures such as refresh token rotation and sender constraint.

Even the new guidance doesn't entirely eliminate some of the inherent risks and complications of implementing an OAuth2 client in a user agent. For example, both access tokens and refresh tokens end up in the user agent, where they are vulnerable to many important attacks; and in general, implementing a full fledged OAuth2 client requires many moving parts (connecting to the appropriate authorization server endpoints, managing communications, handling complex error situations, etc) that can be alleviated but cannot be completely made transparent to the frontend developer by the use of SDKs.

In the attempt to avoid those limitations, developers are increasingly pursuing approaches where their backend components (when available) play a more active role. For example, there are many solutions where the backend takes care of obtaining tokens from the authorization server, using classic confidential client grants, and provides a facade for every API the frontend needs to invoke: in that way, the frontend can simply call the API via facade, securing communications with its backend using mainstream methods such as any cookie based web sign on technology. In some literature the aforementioned pattern, where both token acquisition and API invocation is delegated to a backend, is identified by the term BFF - backend for frontend. Other sources use the term BFF to describe any topology where at least some functionality traditionally implemented by the frontent is delegated to the backend. For the sake of clarity, in this document we will use Full BFF to refer to the approach where both token acquisition and API invocation are handled by the backend, and BFF for approaches where the frontend retain the responsibility to implement some functionality (as it is the case for the pattern described in this specification).

Although the Full BFF approach offers better security, by virtue of keep all tokens out of the user agent, it is not always viable. Depending on the number of users and chattiness of the application, routing every API call thru the backend can be expensive in terms of performance, latency, and service tier of the hosting platform in

use; the solution might rely on user agents connecting to API in the same region; the developmnt and hosting stack might not offer a viable product or technology implementing the pattern. For and other reasons, it is increasingly common practice to use a simpler solution: rely on the backend component for obtaining tokens from the authorization server, and sending back to the frontend the resulting access tokens for direct frontend to API communication. As long as the mechanism used for transmitting tokens from the backend to the frontend is secure, the approach is viable: however leaving the details of its implementation to every application and stack developer results in the impossibility to have frontend and backend development stacks to interoperate out of the box. Furthermore, there are a number of security considerations that, if disregarded in the implementation of the pattern, might lead to elevation of privilege attacks and other challenges.

This documents provides detailed guidance on how to implement the pattern in which a frontend component can delegate token acquisition to its backend component. By offering precise guidance on details such as endpoints and messages format for each operation, this specification will allow developers to create and consume off-the-shelf components that will easily interoperate and allow mixing and matching different frontend and backend SDKs, making it possible to author single page apps consuming APIs on arbitrary domains without having to cope with the complexity normally associated to a frontend-only approach. The OAuth2 for Browser-Based Apps BCP [BrowserBCP] hints at both Full BFF and the approach described here, but doesn't provide detailed guidance - this specification doesn't replace the BCP, it just providers more details to facilitate interoperable and well designed implementations. It's important to stress that the approach here described should be considered only when a Full BFF approach is not viable. Whenever it is possible for a solution to keep tokens out of a user agent, a Full BFF approach should be preferred.

Given that the pattern described here does not provide any artifact that the frontend can use to obtain session information such as user attributes, something traditional approaches to user agent apps development do afford, this document also provides a mechanism for the frontend to obtain session information from the backend.

1.1. Topology and Roles

This document describes how a single page application featuring a backend can obtain tokens from an OAuth2 authorization server to access a resource server. For what the protocol flow is concerned, the topology can be broken down into four roles:

Frontend:

This represents the application code executing in the user agent, controlling presentation and invoking one or more resource servers.

Backend: The backed represents code executing on a server, in particular on the same domain from where the frontend code has been served. Backend and frontend are both under the control of the same developer.

Resource Server: This represents a classic OAuth2 resource server as described in Section 1.1 of OAuth2 [RFC6749], exposing the API the frontend needs to invoke. See Section 6 for more details applying to notable cases.

Authorization Server: This represents a classic OAuth2 authorization server as described in Section 1.1 of OAuth2 [RFC6749], handling authorization for the API the frontend needs to invoke. This document does not introduce any changes in the standard authorization server behavior, however see Section 6 for some security considerations that might influence the policies of individual servers.

1.2. Protocol Flow

This section provides a high level description of the way in which the frontend can obtain and use access tokens with the help of its backend. As a prerequisite for the flow described below, the backend MUST have established a secure session with the user agent, so that all requests from that user agent toward the backend occur over HTTPS and carry a valid session artifact (such as a cookie) that the backend can validate. This document does not mandate any specific mechanism to establish and maintain that session. In other words: the user must have signed in the backend, using whatever web sign on mechanism the developer chooses. For example, the user might have signed in the backend using OpenID Connect [OIDC], resulting in a session cookie bound to the backend application domain that will be included in every future requests from the user agent. The choice of web sign on technology is completely arbitrary, with the only requirement of resulting in an authenticated session.

A second prerequisite establishes that the backend must obtain the access tokens that will be requested by the frontend later on, and or whatever mechanism will allow the backend to renew access tokens (or obtain new ones) in non-interactive fashion. For example, the backend might perform an OAuth2 authorization code flow after sign in to obtain access tokens and refresh tokens; or might have performed an OpenID Connect hybrid flow, satisfying both the sign in and access token acquisition requirements in a single step. Once the backend obtains the tokens, it should persist them in preparation to

hand them over to the frontend when requested to do so, following the flow described below.

[[TODO SVG maybe someday...]]

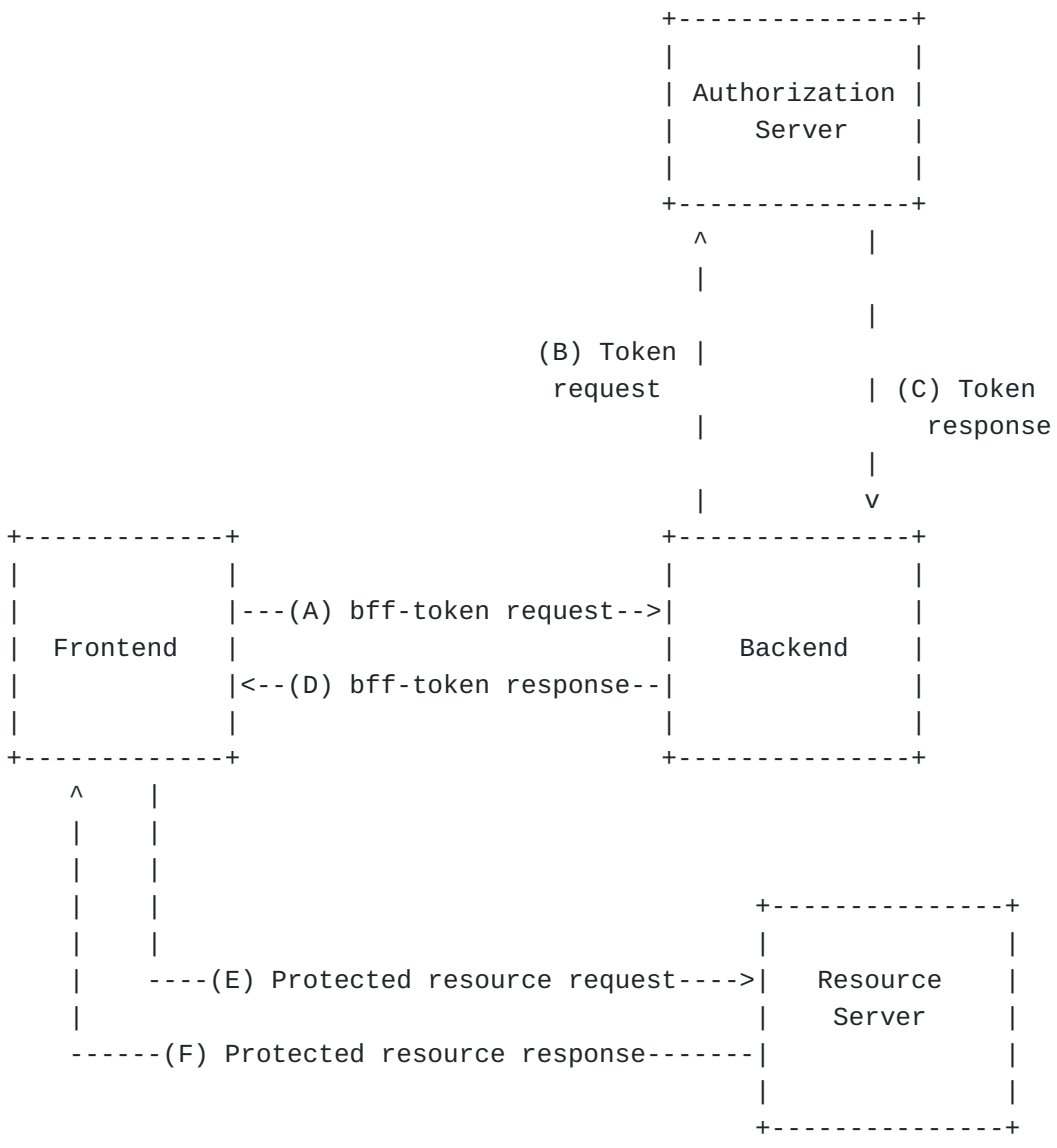

Figure 1: An abstract diagram of the flow followed to obtain an access token and access a protected resource

*(A) The frontend presents to the backend a request for an access token for a given resource server

*(B) If the backend does not already have a suitable access token obtained in previous flows and cached, it requests to the authorization server a new access token with the required characteristics, using any artifacts previousy obtained (eg

refresh token) and grants that will allow the authorization server to issue the requested token without requiring user interaction.

* (C) The authorization server returns the requested token and any additional information according to the grant used (eg validity, actual scopes granted, etc)

* (D) The backend returns the requested access token to the frontend

* (E) The frontend presents the access token to the resource server

* (F) the resource server validates the incoming token and returns the protected resource

2. Conventions and Definitions

The key words "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED", "NOT RECOMMENDED", "MAY", and "OPTIONAL" in this document are to be interpreted as described in BCP 14 [RFC2119] [RFC8174] when, and only when, they appear in all capitals, as shown here.

3. Endpoints

This specification introduces bff-token and bff-sessioninfo, two specialized endpoints that the backend exposes to support the frontend in acquiring tokens and user session information. For the purpose of facilitating the implementation of the pattern with minimal configuration requirements, these endpoints are published at a ".well-known" location according to RFC 5785 [RFC5785]. Both endpoints are meant to be used by the applications' frontend, and the frontend only. As such, the backend MUST verify the the call is occurring in the context of a secure session (e.g., by mandating the presence of a valid session cookie received via HTTPS). The content returned from these two endpoints contains credentials and other sensitive information so MUST also be protected against cross-origin reading of the response data. Preventing successful cross-origin requests in the first place is a strong protection against against cross-origin reads. As such, the endpoints MUST NOT be accessible via CORS and SHOULD have protections in place to prevent Cross-Site Request Forgery. If a cookie is used to maintain the secure session, it SHOULD be marked with HttpOnly [RFC6265] and SameSite [I-D.ietf-httpbis-rfc6265bis]. Both endpoints return JSON [RFC8259] so the response MUST contain a Content-Type header with the correct application/json value and SHOULD also contain a X-Content-Type-Options header with a value of nosniff. Additional guidance around preventing unauthorized reading of response data can be found in

[Post-Spectre-Web-Dev] where the discussion of Dynamic Subresources is particularly relevant.

3.1. The bff-token Endpoint

The bff-token endpoint is exposed by the backend to allow the frontend to request access tokens. It is exposed at the well-known relative URI /.well-known/bff-token. The bff-token endpoint URI MUST use the https scheme. The backend MUST support the use of the HTTP GET method for the bff-token endpoint and MAY support the use of the POST method as well. The backend MUST ignore unrecognized request parameters. See Section 4 for more details on how to use the bff-token endpoint.

3.2. The bff-sessioninfo Endpoint

The bff-sessioninfo endpoint is exposed by the backend to allow the frontend to obtain information about the current user, so that it can be accessed my the presentation code.
It is exposed at the well-known relative URI /.well-known/bff-sessioninfo. The backend MUST support the use of the HTTP GET method for the bff-sessioninfo endpoint. The backend MUST ignore unrecognized request parameters. See Section 5 for more details on how to use the bff-sessioninfo endpoint.

4. Requesting Access Tokens to the Backend

To obtain an access token, the frontend makes a request to the backend at the bff-token endpoint URI. The flow includes the following steps, as shown in Figure 1.

[[TODO more granular error refs]]

1. The frontend generates the request and sends it to the bff-token endpoint as described in Section 4.1 (leg A in Figure 1).

2. The backend examines the request, validating whether it includes a valid user session: if it doesn't, it rejects the request as described in Section 4.3.1.

3. The backend extracts user information from the session, using whatever mechanism it deems suitable, and verifies whether it already has in storage a suitable access token satisfying the request (see Section 6 for more details). If it does, it returns it as described in (5).

4. If there is no suitable access token stored, the backend verifies whether it has the necessary artifacts to request it to the authorization server without requiring user interaction- for example, by using a refresh token previously stored for the

current user. If it does, the backed contacts the authorization server with a token request using the grant of choice (leg B in Figure 1). In the absence of a suitable artifact required to perform a request toward the authorization server, the backend returns an error to the frontend as described in Section 4.3.2.

5. If the authorization server returns the requested token as expected (leg C in Figure 1), the backend returns it to the frontend, as shown in Leg D of Figure 1 and described in Section 4.2. If the authorization server denies the request, the backend returns an error to the frontend as described in Section 4.3.3.

The following sections provide more details for each of the messages described.

4.1. Access Token Request

The frontend requests an access token from the backend by specifying the requirements the resulting token must meet. To do so, the following parameters may be added to to the query component (or request payload in the case of 'POST') of the /.well-known/bff-token request URI using the application/x-www-form-urlencoded format with a character encoding of UTF-8 as described in Appendix B of [RFC6749].

resource : The identifier of the desired resource server, as defined in [RFC8707]. This parameter is OPTIONAL.

scope : The scope of the access request resulting in the desired access token. This parameter follows the syntax described in section 3.3 of [RFC6749]. This parameter is OPTIONAL.

Both parameters MAY be absent from the request. Given that the frontend and the backend are components of the same application, it is possible in some scenarios for the backend to determine what token to return to the frontend without any specific requirement. For example, the application might be consuming only one resource, with a fixed set of scopes: that would make specifying that information in the request from the frontend unnecessary.

The following is an example of request where both resource and scopes are specified.

```
GET /.well-known/bff-token?scope=buy+sell
  &resource=https%3A%2F%2Fapi.example.org%2Fstocks HTTP/1.1
Host: myapp.example.com
Cookie: super-secure-session=hVQvkyX2IOj36fqIoUQFlBeALbh
```

Note that the request does not need to specify any client attributes, as those are all handled by the backend- and the presence of a pre-existing session provides the context necessary for the backend to select the right settings when crafting requests for the authorization server.

4.2. Access Token Response

If the backend successfully obtains a suitable token, or has one already cached, it returns it to the frontend with the following parameters in the payload of the HTTP response using the application/json media type as defined by [RFC8259].

access_token : The requested access token. This parameter is REQUIRED.

expires_in : The lifetime in seconds of the access token, as defined in section 5.1 of [RFC6749]. This parameter is REQUIRED, if the information was made available from the authorization server that originally issued the access token.

scope : The scope of the access token being returned as list of space-delimited, case-sensitive strings, as defined in Section 3.3 of [RFC6749]. If the request contained a scope parameter, and the scope of resulting token is different from the requested value, this parameter is REQUIRED. In all other cases, the presence of scope in the response is OPTIONAL.

The following is an example of access token response.

```
HTTP/1.1 200 OK
Content-Type: application/json
X-Content-Type-Options: nosniff
Cache-Control: no-cache, no-store
Cross-Origin-Resource-Policy: same-origin
Content-Security-Policy: sandbox
Cross-Origin-Opener-Policy: same-origin
X-Frame-Options: DENY

{
  "access_token":"4bWc0ESC9aCc77LTC8EjR1pCfE4WxfNg",
  "expires_in":3596,
  "scope":"buy sell"
}
```

Note that if the backend elects to cache tokens, to serve future requests from the frontend without contacting the authorization server if still within the useful lifetime, it must also cache expiration information and scopes in accordance to the requirements expressed in this section.

4.3. Errors

When the backend fails to deliver to the frontend the requested token, it responds with an HTTP 400 (Bad Request) status code and includes the following parameters with the response:

error : An ASCII error code identifying the circumstances of the error. See the next sections for details. This parameter is REQUIRED.

error_description : OPTIONAL. A human-readable message describing the error for troubleshooting purposes.

4.3.1. No valid session found

All requests to the backend MUST be performed in the context of a valid authenticated session, typically by presenting a session cookie over a TLS channel. If the backend cannot find or validate a session, it must reject the request and return a message as described in Section 4.3, with an error parameter value of invalid_session.

4.3.2. Backend cannot perform a request to the authorization server

If the backend doesn't have the necessary artifacts (e.g., a refresh token for the current user and/or requested resource) to request a suitable access token to the authorization server without requiring user interaction, it will reject the request and return a message as described in Section 4.3, with an error parameter value of backend_not_ready.

4.3.3. The backend request to the authorization server fails

If the backend request to the authorization server fails, the backend will return to the frontend a message as described in Section 4.3, with as error parameter value the error parameter received in the authorization server response (as described by section 5.2 of [RFC6749] and, if present in the authorizations server response, will include the error_description parameter with the same parameter value as received by the authorization server. [[TODO wow this sentence is ugly.]]

5. Requesting Session Information from the Backend

Application developers will often need to obtain information about the current session (such as user attributes, session expiration, etc) to display it to the end user, drive application behavior and any other operation it would perform if the frontend would be in charge of obtaining tokens directly. In the topology described in this specification, most of the user experience is driven by the

frontend: however, the session information is inaccessible to the user agent, as it is either kept in artifacts that the user agent cannot inspect (opaque sessions cookies) or on the backend side. The /.well-known/bff-sessioninfo endpoint is meant to restore the developer's ability to access the session information they need, without compromising the security of the solution. At any time, the frontend can leverage the current secure session to send to the bff-sessioninfo endpoint a request, and receive the needed session information. The following sections provide details on request and response messages.

5.1. Session Information Request

The frontend sends a request for session information via an HTTP GET, using the https scheme in the context of a secure session. The request has no parameters. The following is an example of session information request.

```
GET /.well-known/bff-sessioninfo HTTP/1.1
Host: myapp.example.com
Cookie: super-secure-session=hVQvkyX2IOj36fqIoUQFlBeALbh
```

5.2. Session Information Response

If the request is executed in the context of a secure session, the backend returns a JSON object containing any information it deems appropriate to share with the frontend about the content of the session. For example, if the session was established via OpenID Connect [OIDC] the response might contain the session and user attribute claims as defined in sections 2 and 5.1 of [OIDC]. The following is a non-normative example of such a session information response.

```
HTTP/1.1 200 OK
Content-Type: application/json
X-Content-Type-Options: nosniff
Cache-Control: no-cache, no-store

{
    "iss": "https://as.example.com",
    "sub": "24400320",
    "exp": 1311281970,
    "auth_time": 1311280969,
    "preferred_username": "johnny",
    "email_verified: "johnny@foo.com",
    "given_name": "Jonathan",
    "family_name" : "Swift"
}
```

It is worth noting that the backend isn't bound to any specific rule and is free to return any information it deems necessary in this message in the context of the application (frontend and backend) own requirements.

5.3. Error

In case the frontend sends a request to bff-sessioninfo in the absence of a valid secure session, the backend will return an error as described in Section 4.3.2. For any other error situation, the backend is free to determine what to signal to the frontend. [[TODO seems a bit weak... maybe a generic error?]]

6. Security Considerations

The simplicity of the described pattern notwithstanding, there are a number of important considerations that frontend, backend and SDK implementers should keep in mind while implementing this approach.

6.1. Frontend should not persist access tokens in local storage

Access tokens SHOULD NOT be saved in local storage: they SHOULD be kept in memory, and retrieved anew when necessary from the backend following Section 4.

6.2. Mismatch between security characteristics of token requestor and API caller

Some authorization servers might express in their access tokens whether the client obtaining it authenticated itself, or it behaved as a public client. Resource servers might rely on that information to infer the nature and security characteristics of the application presenting the access token to them, and use that to drive authorization decisions (e.g., only allow certain operations if the caller is a confidential client). The pattern described here obtains an access token through the backend, a confidential client, but the access token is ultimately used by code executing in a far less secure environment. Resource servers knowing that their clients will use this pattern SHOULD refrain from using the client authentication type as a factor in authorization decision, or, whenever possible, should use whatever extensions the authorization server of choice offers to signal that the requested access tokens will not be used by a confidential client. As there are no standards to express in an access token the nature of the client authentication used in obtaining the token itself, this document does not provide a specific mechanism to influence the authorization server and leaves the task, in the rare cases it might be necessary, to individual implementations.

6.3. Mismatch between scopes in a request vs cached tokens

The backend will likely cache token responses from the authorization server, so that the backend can promptly serve equivalent requests from the frontend without further roundtrips toward the authorization server. That is a powerful optimization, but it presents scopes elevation risks if applied indiscriminately. If the token cached by the authorization server features a superset of the scopes requested by the frontend, the backend SHOULD NOT return it to the frontend and perform a new request with the smaller scopes set to the authorization server.

6.4. Resource server colocated with the backend

If the only API invoked by the frontend happens to be colocated with the backend, the frontend doesn't need to obtain access tokens to it: it can simply use the same secure session leveraged to protect requests to the token endpoints described here. The bff-token isn't necessary in that scenario, although bff-sessioninfo retains its usefulness to surface session and user information to the user agent code. Also note that the presence of the bff-token endpoint makes it possible to easily accommodate possible future evolutions where the frontend needs to invoke APIs protected by resource servers hosted elsewhere, without engendering changes in the security property of the application. Developers choosing to expose API

7. IANA Considerations

This specification requests registration of the following two well-known URIs in the IANA "Well-Known URIs" registry [IANA.well-known] established by [RFC5785].

The bff-token Endpoint

- *URI suffix: bff-token
- *Change Controller: IESG
- *Specification Document: Section 3.1 [[of this specification]]
- *Related information: (none)

The bff-sessioninfo Endpoint

- *URI suffix: bff-sessioninfo
- *Change Controller: IESG
- *Specification Document: Section 3.2 [[of this specification]]

*Related information: (none)

Miscellaneous

[[TODO Should we say something about: Requests could be more complicated than just scopes (think RAR) and the frontend might need to tell more than scopes to the backend. In that case, just add custom params and stir.]]

[[TODO We mentioned another thing, but I can't remember now.]]

8. Normative References

[RFC5785] Nottingham, M. and E. Hammer-Lahav, "Defining Well-Known Uniform Resource Identifiers (URIs)", RFC 5785, DOI 10.17487/RFC5785, April 2010, <https://www.rfc-editor.org/info/rfc5785>.

[RFC6749] Hardt, D., Ed., "The OAuth 2.0 Authorization Framework", RFC 6749, DOI 10.17487/RFC6749, October 2012, <https://www.rfc-editor.org/info/rfc6749>.

[RFC8259] Bray, T., Ed., "The JavaScript Object Notation (JSON) Data Interchange Format", STD 90, RFC 8259, DOI 10.17487/RFC8259, December 2017, <https://www.rfc-editor.org/info/rfc8259>.

[RFC6750] Jones, M. and D. Hardt, "The OAuth 2.0 Authorization Framework: Bearer Token Usage", RFC 6750, DOI 10.17487/RFC6750, October 2012, <https://www.rfc-editor.org/info/rfc6750>.

9. Informative References

[BrowserBCP] Parecki, A. and D. Waite, "OAuth 2.0 for Browser-Based Apps", 5 April 2021, <https://tools.ietf.org/html/draft-ietf-oauth-browser-based-apps-07>.

[RFC2119] Bradner, S., "Key words for use in RFCs to Indicate Requirement Levels", BCP 14, RFC 2119, DOI 10.17487/RFC2119, March 1997, <https://www.rfc-editor.org/info/rfc2119>.

[RFC8174] Leiba, B., "Ambiguity of Uppercase vs Lowercase in RFC 2119 Key Words", BCP 14, RFC 8174, DOI 10.17487/RFC8174, May 2017, <https://www.rfc-editor.org/info/rfc8174>.

[RFC6265] Barth, A., "HTTP State Management Mechanism", RFC 6265, DOI 10.17487/RFC6265, April 2011, <https://www.rfc-editor.org/info/rfc6265>.

[RFC8707] Campbell, B., Bradley, J., and H. Tschofenig, "Resource Indicators for OAuth 2.0", RFC 8707, DOI 10.17487/RFC8707, February 2020, <https://www.rfc-editor.org/info/rfc8707>.

[I-D.ietf-oauth-security-topics] Lodderstedt, T., Bradley, J., Labunets, A., and D. Fett, "OAuth 2.0 Security Best Current Practice", Work in Progress, Internet-Draft, draft-ietf-oauth-security-topics-16, 5 October 2020, <https://tools.ietf.org/html/draft-ietf-oauth-security-topics-16>.

[OIDC] Sakimura, N., Bradley, J., Jones, M., de Medeiros, B., and C. Mortimore, "OpenID Connect Core 1.0 incorporating errata set 1", 8 November 2014, <http://openid.net/specs/openid-connect-core-1_0.html>.

[I-D.ietf-httpbis-rfc6265bis] West, M. and J. Wilander, "Cookies: HTTP State Management Mechanism", Work in Progress, Internet-Draft, draft-ietf-httpbis-rfc6265bis-07, 7 December 2020, <https://tools.ietf.org/html/draft-ietf-httpbis-rfc6265bis-07>.

[I-D.ietf-oauth-v2-1] Hardt, D., Parecki, A., and T. Lodderstedt, "The OAuth 2.1 Authorization Framework", Work in Progress, Internet-Draft, draft-ietf-oauth-v2-1-00, 30 July 2020, <https://tools.ietf.org/html/draft-ietf-oauth-v2-1-00>.

[RFC7636] Sakimura, N., Ed., Bradley, J., and N. Agarwal, "Proof Key for Code Exchange by OAuth Public Clients", RFC 7636, DOI 10.17487/RFC7636, September 2015, <https://www.rfc-editor.org/info/rfc7636>.

[Post-Spectre-Web-Dev] West, M., "Post-Spectre Web Development (work in progress)", 16 March 2021, <https://www.w3.org/TR/2021/WD-post-spectre-webdev-20210316/>.

[IANA.well-known] IANA, "Well-Known URIs", <https://www.iana.org/assignments/well-known-uris>.

Appendix A. Acknowledgements

I wanted to thank the Academy, the viewers at home, etc..

Appendix B. Document History

[[To be removed from the final specification]]

-01

*Added some protections/discussions around CSRF and cross-site reading of sensitive data.

-00

*Literally willed into existence by a long haired gentleman from the Seattle area

Authors' Addresses

Vittorio Bertocci
auth0.com

Email: vittorio@auth0.com

Brian Campbell
Ping Identity

Email: bcampbell@pingidentity.com