Whitepaper

The Backend for Frontend pattern

Learn how to keep tokens more secure by using the Backend for Frontend (BFF) architectural pattern.



auth0

Table of Contents

2 Introduction

3 OAuth 2.0 Client Types and Security

4 SPA Security and Tokens

5 The Backend For Frontend Pattern

10 Caveats and Variations

11 Summary



Introduction

One of the biggest challenges for applications using OAuth 2.0 and OpenID Connect is token security. Preventing an ID, access, or refresh token from falling into the wrong hands is a priority of these protocols. The specs have adopted several technical solutions, and some best practices have been proposed. Among the latter is the use of the Backend for Frontend pattern, whose rationale and architecture we will analyze in this article.



Andrea Chiarelli
Principal Developer Advocate

OAuth 2.0 Client Types and Security

OAuth 2.0 defines two types of client applications: public and confidential clients. This classification is based on the application's ability to keep authentication credentials secure. From the specs:

- **Confidential.** Clients capable of maintaining the confidentiality of their credentials (e.g., client implemented on a secure server with restricted access to the client credentials), or capable of secure client authentication using other means.
- **Public.** Clients incapable of maintaining the confidentiality of their credentials (e.g., clients executing on the device used by the resource owner, such as an installed native application or a web browser-based application), and incapable of secure client authentication via any other means.

In a nutshell, confidential clients are those applications that run in a protected environment, such as a server, while public clients are applications that run in an uncontrolled environment, such as the user's device. Typical confidential clients are traditional web applications, while typical public clients are Single-Page Applications (SPA), desktop and mobile apps.

From a security perspective, public clients can't store their credentials securely. There is a high risk of tokens falling into the wrong hands. This can happen either because of a potential lack of protection on the local system or because of the potentially large number of instances of the application that increase the attack surface.

To mitigate the problem of issuing a token to a client that can't be authenticated (i.e., a public client), a specific flow has been designed: the Authorization Code Flow with PKCE. Anyway, this does not solve all the concerns about the security of a public client.

SPA Security and Tokens

Another concern for public clients is token storage. Once an ID or access token is received from the server, how can a public client protect it from unauthorized access? As long as the application is running, the application memory is the safest place to store tokens. But what if the application stops running temporarily?

For example, consider a SPA that has a token in memory. What happens when the user refreshes the page on which the application is running? The token is lost, and the user is forced to authenticate again for the application to obtain a new token.

You could consider storing the token in the browser's local storage or in a cookie and reloading it after the refresh, but this is not a recommended practice. A malicious user or application could gain access to the local storage or cookie and obtain the token. See this article for more information on the risks of using browser storage.

Something similar happens with native applications (desktop and mobile): when the user closes the application, all tokens are lost, requiring authentication the next time the application is opened. While this may be acceptable in certain contexts, many users have become accustomed to not having to authenticate every time they open their app, especially in the mobile environment. However, native applications have an advantage over SPAs. The presence of secure storage mechanisms in most popular operating systems, such as Keychain in iOS/Mac or Keystore in Android.

The level of security required by an application depends on its specific functionality. It is the result of balancing the data confidentiality requirements with the ease of implementation. Therefore, in some contexts, using the Authorization Code Flow with PKCE and all the concerns about token management in a SPA may be acceptable. In other contexts, it is not. What can you do for those contexts where you can't absolutely risk a token falling into the wrong hands?

The Backend For Frontend Pattern

A seemingly marginal note in the definition of client in the OAuth 2.0 specification suggests a solution:

A client may be implemented as *a distributed set of components, each with a different client type and security context* *(e.g., a distributed client with both a confidential server-based component and a public browser-based component).*

This paves the way for the adoption of a well-known architectural pattern: the **Backend for Frontend** (BFF) pattern. This is a popular pattern to address specific needs for a client platform by building a dedicated backend. The needs may be of various types: decoupling of concerns, flexibility and maintainability, optimized performance, security, etc.



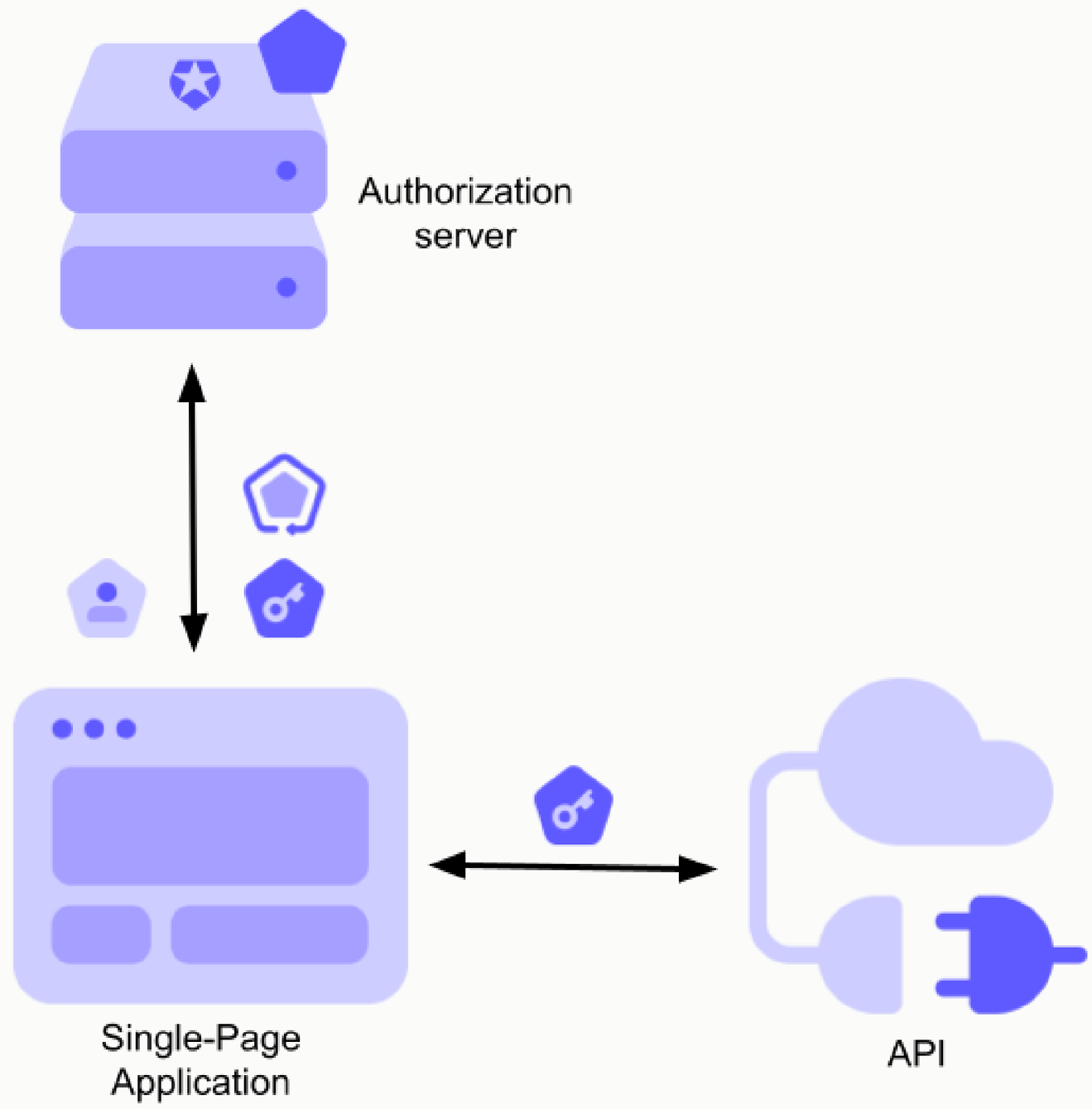
Read this document to learn how this architectural pattern was born.

In our case, we are interested in strengthening the security of SPAs in the context of OIDC and OAuth 2.0, and the best current practices give us some guidelines about using this pattern.

Let's explore how the BFF pattern works.

The BFF architecture

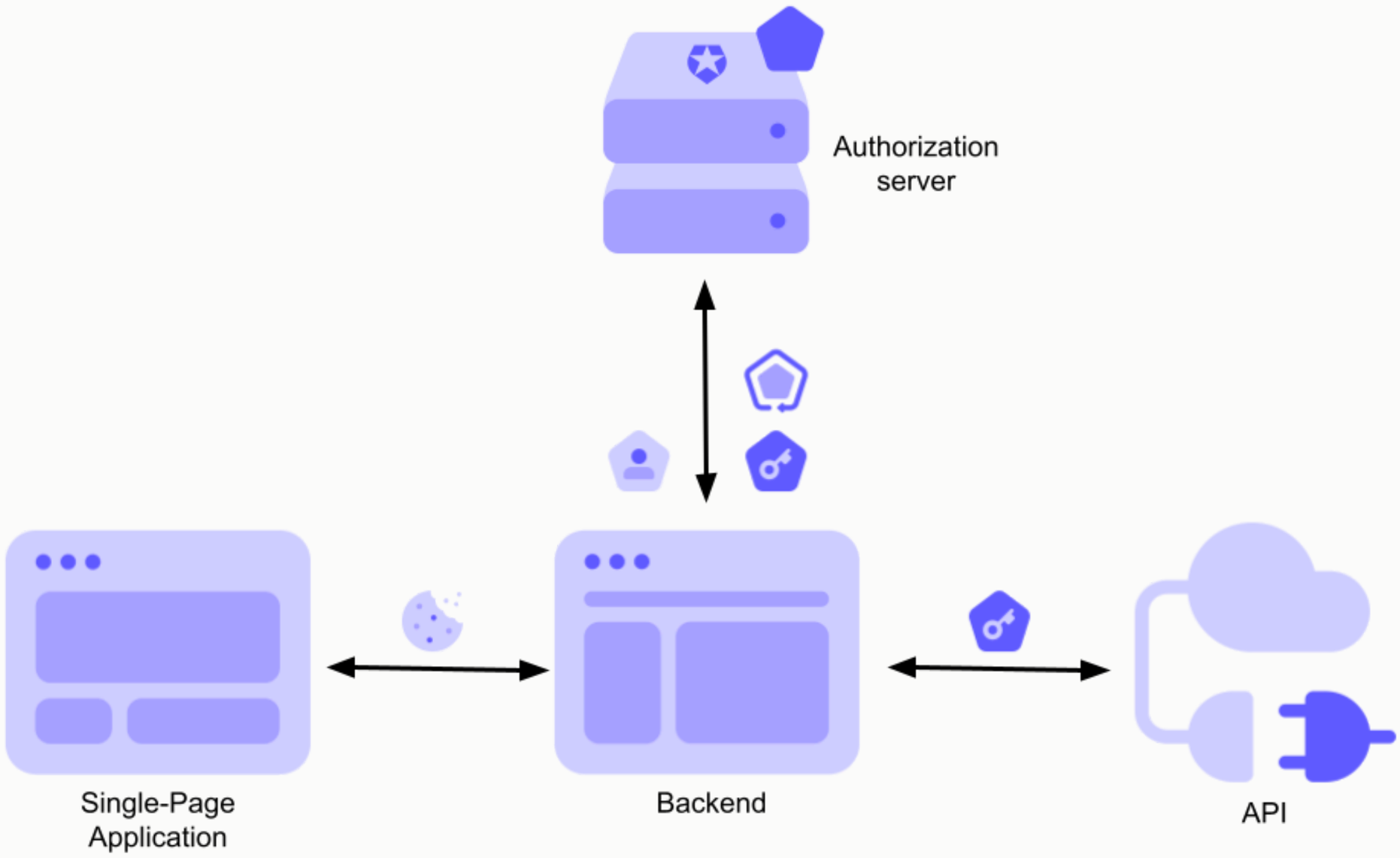
The actors involved in a typical scenario where a SPA uses the Authorization Code Flow with PKCE are shown in the following diagram:



The SPA interacts with the authorization server to get the ID, access, and refresh tokens. Then, the SPA uses the ID token to get data about the user, the access token to call an API, and the refresh token to get a new access token once it expires.

In this scenario, all the tokens are in the hands of the SPA.

By applying the BFF pattern, we are going to enter a new component in this architecture, the dedicated backend, as shown below:



In this new scenario, the backend has three core responsibilities:

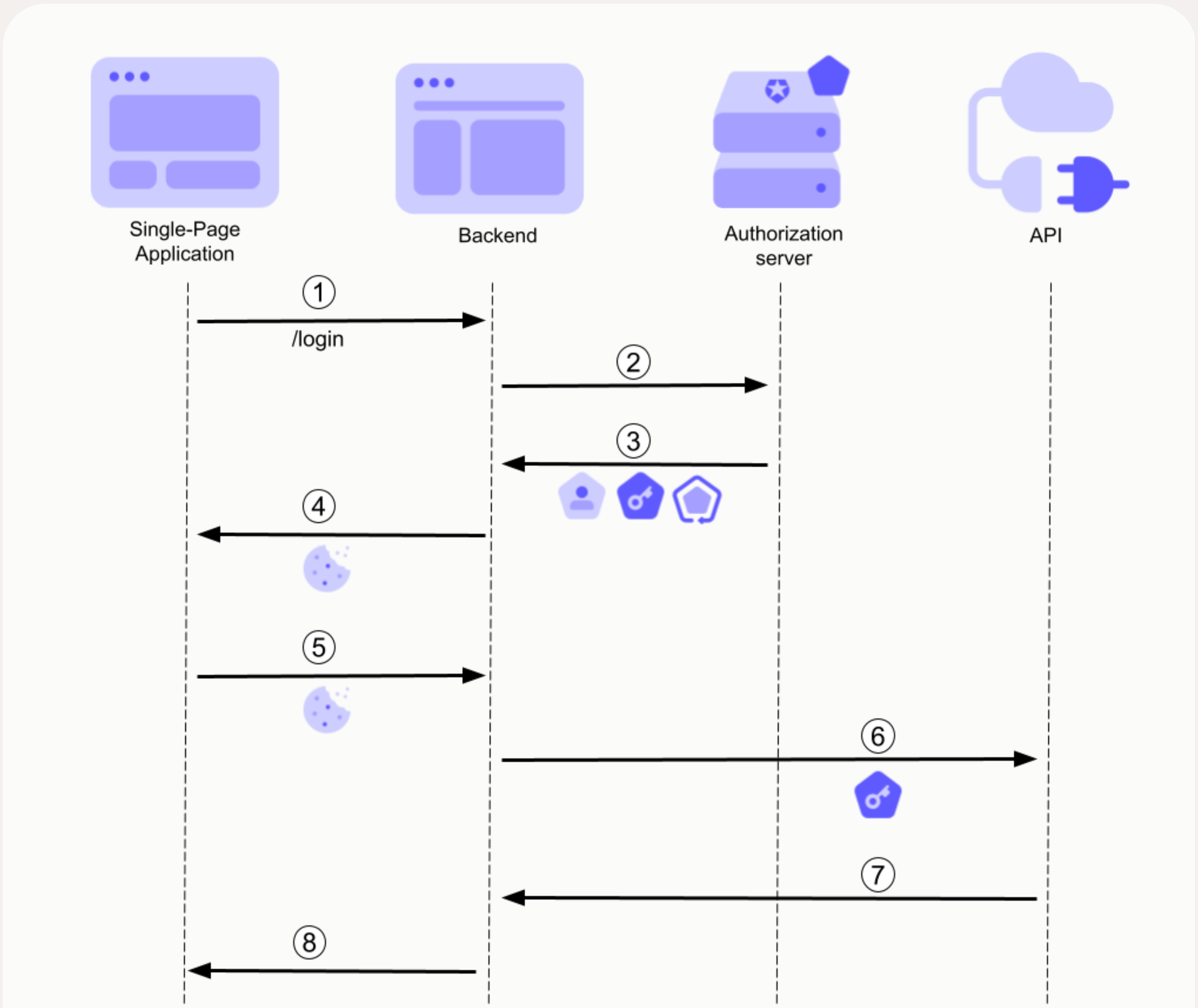
- It interacts with the authorization server as a confidential client.
- It manages all the tokens on behalf of the SPA, which can't access them anymore.
- It proxies all requests to the API, embedding the access token before forwarding them.

The SPA only interacts with the backend, relying on traditional cookies for authenticated sessions. Basically, the backend exposes a number of endpoints that let the SPA access the user profile data and the remote API.

This architecture improves the security of token negotiation, since it now happens via a confidential client, and token storage, since it now occurs on the server side.

The BFF flow

Let's take a look at how the actors in this architecture interact with each other by following this diagram:



Here are the descriptions for each step:

1. The SPA points to a backend URL (`/login` in the example above) to let the user authenticate.
2. The backend starts the usual OpenID Connect negotiation with the authorization server: it redirects the user browser to the authorization endpoint to authenticate the user and get the needed tokens.
3. The authorization server issues the tokens to the backend, which can store it in a cache.
4. The backend issues a cookie representing the user session and triggers the reloading of the SPA in the browser.
5. The SPA makes an API request to the backend including the cookie.
6. The backend validates the SPA request, retrieves the associated access token, and uses the token to make a call to the API.
7. The API returns a response to the backend.
8. The backend forwards the response to the SPA.

Caveats and Variations

The BFF pattern solves the main concerns about using SPAs in an OIDC/OAuth 2.0 flow:

- There is no longer a public client negotiating the tokens, but this negotiation is now the responsibility of a confidential client represented by the backend.
- The SPA no longer needs to store tokens to prevent user re-authentication after page refresh.

To properly implement the BFF, there are a few caveats you need to pay attention to:

- The session cookie issued by the backend must be marked `Secure` and `HttpOnly`.
- The cookie must be associated with the token(s) issued by the authorization server.
- The tokens can be stored on the server side or in the session cookie. In the latter case, the cookie must be encrypted.
- The backend must implement all the measures to prevent Cross-Site Request Forgery (CSRF) attacks.
- The BFF pattern does not work for a standalone SPA that relies on calling a protected API directly from JavaScript. The SPA must be hosted by the backend, which is implemented as a regular web application.

While the BFF pattern solves the main security concerns of token exchange and storage for SPAs, some developers are concerned about performance issues. Specifically, the mediation run by the backend as a proxy API can be a bottleneck in some contexts. There is an alternative to the BFF pattern that meets this need, but it comes at the cost of reduced security.

This alternative is the Token-Mediating Backend pattern, which allows the backend to negotiate the tokens as in the BFF pattern but provides the access token to the SPA. This way, the SPA can directly call the protected API using the access token. While this pattern keeps token negotiation secure by always relying on the backend as a confidential client, it leaves the issue of storing and protecting the access token open. To overcome this issue, you can consider using OAuth 2.0 with DPoP, which binds an access token to the client, making it no longer a bearer token. In this case, you need a DPoP-enabled authorization server, of course.

Summary

Now you have a better understanding of the concerns related to token management and SPAs. You learned that SPAs, like any public client, are not always a safe place to handle ID, access, and refresh tokens. Much depends on the specific business requirements of the application, of course.

If you need a more secure architecture for your SPA, you can apply the Backend for Frontend pattern. This pattern introduces a dedicated backend for token negotiation and management on behalf of the SPA. This way, no tokens will reach the SPA, while the backend will also take charge of proxying the requests to a protected API.