OpenID Connect IN ACTION

Prabath Siriwardena

MEAP

MANNING

MEAP Edition
Manning Early Access Program
OpenID Connect in Action
Version 6

For more information on this and other Manning titles go to
manning.com

3 Securing access to a single-page application

This chapter covers

- OpenID Connect authentication flows and how they differ from OAuth 2.0 grant types
- How implicit authentication flow works with a single-page application
- How authorization code flow works with a single-page application
- Why you need to pick authorization code flow over implicit flow
- Securing a React-based single-page application with OpenID Connect

With the heavy adoption of APIs, over time, single-page applications (SPA) have become one of the most popular options for building client applications on the web. If you are new to single-page application architecture, we recommend you first go through the book SPA Design and Architecture: Understanding Single Page Web Applications by Emmit Scott (Manning Publication, 2015). Also in this chapter we assume you have a good knowledge of OAuth 2.0, which is the fundamental building block of OpenID Connect. In case you don't, please go through chapter 2 before following the rest of the chapter.

Under OAuth 2.0 terminology, a SPA is identified as a public client application.[1] In principle, a public client application is unable to hide any secrets from the users of it. Most of the time a SPA is an application written in JavaScript that runs on the browser; so, anything on the browser is visible to the users of that application. This is a key-deciding factor on how you want to use OpenID Connect to secure a SPA.

In this chapter we'll teach you what OpenID Connect authentication flows are and how different OpenID Connect authentication flows work with a SPA. You will also learn how to build a SPA using React and then log in to it via OpenID Connect. React is the most popular

[1] In chapter 2, under the section 2.4 we discuss OAuth 2.0 client types and their characteristics.

JavaScript library for developing user interfaces. If you are new to React, please go through appendix A first.

3.1 Authentication flows define the communications between a client application and an OpenID provider

In this section you'll learn what is an authentication flow in OpenID Connect and different types of authentication flows.

In chapter 1, you learnt that OpenID Connect defines a schema for a token (which is a JSON Web Token (JWT)) to exchange information between an OpenID provider and a client application; and a set of processing rules around it. The OpenID Connect specification identifies this token, as the ID token, which we will briefly discuss in this chapter and in detail in chapter 4.

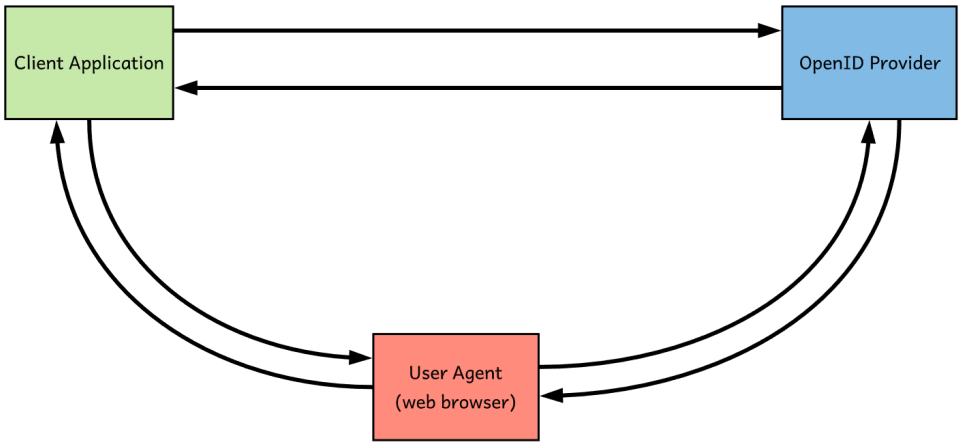

Figure 3.1: OpenID authentication flows define how the client application communicates with the OpenID provider to authenticate an end user. Some communications happen via the web browser and some happens directly between the client application and the OpenID provider.

In addition to the ID token, OpenID Connect specification also introduces a transport binding, which defines how to transport an ID token from an OpenID provider to a client application (figure 3.1). In OpenID Connect, we use the term ***authentication flows*** to define multiple ways by which you can transport an ID token from an OpenID provider to a client application.

OpenID Connect defines three authentication flows:

- ***authorization code*** flow,
- ***implicit*** flow, and
- ***hybrid*** flow.

In section 3.3 you learn how implicit flow works and in section 3.9 how authorization code flow works. We'll discuss hybrid flow in detail in chapter 6.

3.2 Authentication flows vs. grant types

The OAuth 2.0 core specification (RFC 6749) introduced four grant types, which we discussed in chapter 2 in detail[2]. In this section you'll learn how an OpenID Connect authentication flow relates to a grant type as well as the differences.

A grant type in OAuth 2.0 defines a protocol how a client application can obtain an access token from an authorization server. Typically, a grant type defines four key components (please see section 2.3 for the details): ***authorization request***, ***authorization response***, ***access token request*** and ***access token response***.

An authentication flow in OpenID Connect uses grant types, but an authentication flow is more than a grant type (table 3.1). Typically, an authentication flow in OpenID Connect defines four key components, quite similar to an OAuth 2.0 grant type, but not exactly the same: ***authentication request***, ***authentication response***, ***token request*** and ***token response***. You might have already noticed the differences; in a grant type we have an authorization request/response, while in an authentication flow we have an authentication request/response, also in a grant type we have an access token request/response, while in an authentication flow we have a token/request response.

Table 3.1: The differences in the terminology, OAuth 2.0 vs. OpenID Connect

OAuth 2.0	OpenID Connect
Authorization request: Initiated from the from client application to the authorization server. The scope and redirect_uri are optional parameters in the authorization request.	**Authentication request**: Initiated from the client application to the OpenID provider. The scope and redirect_uri are requred parameters in the authorization request.
Authorization response: Initiated from the authorization server to the client application	**Authorization response**: from the OpenID provider to the client application
Access token request: Initiated from the client application to the authorization server	**ID token request**: from the client application to the OpenID provider
Access token response: Initiated from the client application to the authorization server	**ID token response**: from the OpenID provider to the client application

The OpenID Connect specification defines the authentication flows in a self-contained manner in itself. So we should not confuse the OAuth 2.0 grant types with OpenID Connect authentication flows. The authorization code flow in OpenID Connect is not as same as the authorization code grant type in OAuth 2.0, and the implicit flow in OpenID Connect is not as same as the implicit grant type in OAuth 2.0.

2 The RFC 6749 in fact introduced five grant types. However, the behavior of refresh token grant type is quite different from other four: authorization code, implicit, password and client credentials. When we say OAuth 2.0 defines four grant types, we refer to those four core grant types. As discussed in chapter 2, OAuth 2.1 has removed implicit and password grant types from the specification.

3.3 How does implicit flow work?

In this section you'll learn how an OpenID provider transports an ID token to a client application using the implicit flow. The sequence of events or steps happens during this flow, as well as the messages being passed in each step is clearly defined in the OpenID Connect specification.

3.3.1 The flow of events in the implicit authentication flow

Figure 3.2 shows the sequence of events happens between the OpenID provider, the client application, and the user. The client application in figure 3.2 can be any type of an application, but here our discussion mostly focuses on a SPA. Also, the implicit flow is more popular among SPAs than any other application type. In the following sections we discuss in detail what happens in each step in figure 3.2.

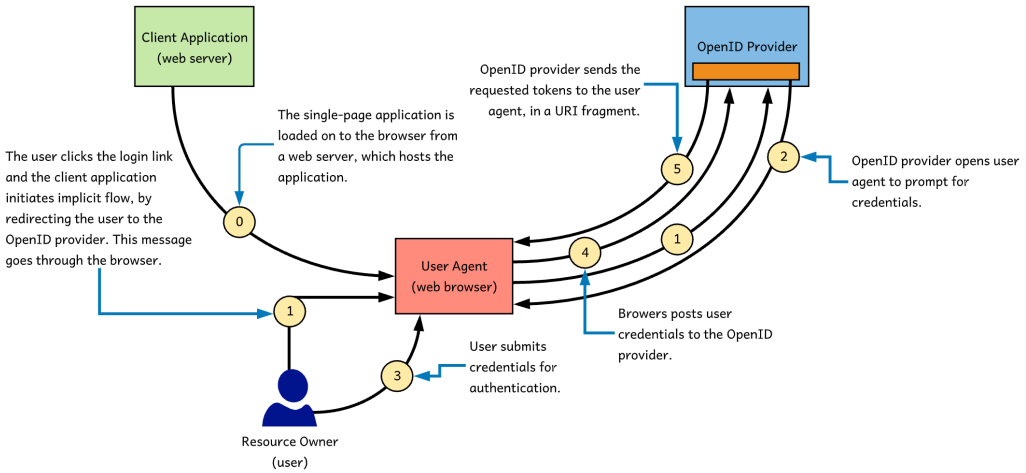

Figure 3.2: The client application uses implicit authentication flow to communicate with the OpenID provider to authenticate the user.

THE CLIENT APPLICATION INITIATES A LOGIN REQUEST VIA THE BROWSER

In the step 1 of figure 3.2, the user clicks on the login link and the client application initiates a login request via the browser. In the case of a SPA, we can expect that the user clicks on a login link on the web page of the client application, and browser does an HTTP GET to the ***authorize*** endpoint of the OpenID provider.

The authorize endpoint of the OpenID provider is a well-known endpoint and the client applications can find it by going through the OpenID provider documentation or else using OpenID Connect discovery protocol, which we discuss in detail in chapter 12. If you use Google as your OpenID provider, then this is the authorize endpoint of Google, which you can find from their documentation: https://accounts.google.com/o/oauth2/v2/auth.

The request the client application generates in step 1 of figure 3.2 is called an ***authentication request***. You may recall from the chapter 2, in OAuth 2.0 the request initiated from the client application to the OAuth 2.0 authorization server is called an ***authorization request***.

The following listing shows an example of an authentication request. This is in fact a URL constructed by the client application, which takes the user to the authorize endpoint of the OpenID provider, when the user clicks on the login link.

Listing 3.1: Authentication request generated by the client application

```
https://accounts.google.com/o/oauth2/v2/auth?
      client_id=424911365001.apps.googleusercontent.com&
      redirect_uri=https%3A//app.example.com/redirect_uri&
      scope=openid email&.   #A
      login_hint=jdoe@example.com&
      response_type=id_token token&.   #B
      state=Xd2u73hgj59435&
      nonce=0394852-3190485-2490358
```

#A The scope values are separated by a space. However, when you type this on the browser, the browser will URL encode the space, so the space will be replaced by %20.

#B The response_type values are separated by a space. However, when you type this on the browser, the browser will URL encode the space, so the space will be replaced by %20.

Let's go through the query parameters added to the authentication request by the client application, as shown in listing 3.1. The definition of these parameters are consistent across all three authentication flows the OpenID Connect defines, however, the values may change.

- ***client_id***: This is an identifier the OpenID provider uses to uniquely identify a client application. The client application gets a `client_id` after registering itself at the OpenID provider. For registration at the OpenID provider, either you can follow an out-of-band mechanism provided by the OpenID provider or use OpenID Connect dynamic client registration API, which we discuss in chapter 12.[3] The `client_id` is a required parameter in the authentication request, and is originally defined in the OAuth 2.0 specification, which we discussed in detail in chapter 2.
- ***redirect_uri***: This is an endpoint belongs to the client application. After successfully authenticating the user and getting the consent from the user to share the requested data with the client application, the OpenID provider redirects the user to the `redirect_uri` endpoint along with the requested tokens (step 5 of figure 3.2). During the client registration process at the OpenID provider, you need to share the exact URI you use for `redirect_uri` parameter in the authentication request, with the OpenID provider.
- The OpenID provider will do one-to-one matching of the value of the `redirect_uri` in the authentication request against the one already registered by the client application. Most OpenID providers do an exact match between these two URIs. However, some OpenID providers let the client applications register multiple URI and some let the client applications define a regular expression for the validation of the `redirect_uri`.

3 A out-of-band mechanism could be a developer registering a client application using the UI provided by the OpenID provider.

- Doing a validation against a regular expression gives more flexibility to dynamically change the redirection path by the client application, but should use consciously and regular expression used for validation must be thoroughly tested. In the section 3.3 we explain the use cases where you want to have multiple `redirect_uri`s for the same client application.
- The `redirect_uri` is a required parameter in the authentication request, and is originally defined in the OAuth 2.0 specification. However, in the OAuth 2.0 specification the `redirect_uri` is not a required parameter for both the implicit and authorization code grant types.
- ***scope***: The value of `scope` parameter is just a string, where both the client application and the OpenID provider should be able to interpret the meaning of it. Any OpenID Connect authentication request must carry the value ***openid*** for the `scope` parameter. You can have multiple values for the `scope` parameter, each separated by space, but one of them must be openid.
- The OpenID Connect specification defines four scope values (profile, email, address and phone) in addition to the `openid` scope. A client application can use any of these scope values to request claims from the OpenID provider. In chapter 5, we discuss in detail how to use scopes to request claims.
- The `scope` is a required parameter in the authentication request, and is originally defined in the OAuth 2.0 specification. However, in the OAuth 2.0 specification the `scope` is not a required parameter for both the implicit and authorization code grant types.
- ***login_hint:*** The value of `login_hint` parameter is a string that carries some hint with respect to the user (or the application), which can be used by the OpenID provider to build a better user experience. For example, if the application already knows the user's email address (probably from a cookie stored under the domain of the client application), then that can go as the value of the `login_hint` parameter and the OpenID provider can directly request the user to share the credentials, rather asking for an user identifier.
- The `login_hint` is an optional parameter introduced by the OpenID Connect specification, which you do not find in the OAuth 2.0 specification. In chapter 11 we discuss a couple of use cases of `login_hint` with respect to OpenID federation.
- ***response_type***: The value of the `response_type` parameter in the authentication request defines which tokens the authorization endpoint of the OpenID provider should return back to the client application.
- In the implicit flow there are two possible values: ***id_token*** or ***id_token token***. If the value of the `response_type` is `id_token`, then the authorization endpoint will only return back an ID token, and if the `response_type` is `id_token token`, then the authorization endpoint will return back an ID token and an access token.
- The `response_type` is a required parameter in the authentication request, and is originally defined in the OAuth 2.0 specification.

- ***response_mode***: The value of the `response_mode` parameter in the authentication request defines how the client application expects the response from the OpenID provider. This is an optional parameter and is not in the listing 2.1. If you set the value of the `response_mode` parameter to ***query***, for example, then all the parameters in the response (from the OpenID provider) are encoded as a query string added to the `redirect_uri` as shown below.

```
https://app.example.com/redirect_uri?token=XXXXX&id_token=YYYYYY
```

- If you set the value of the `response_mode` parameter to ***fragment***, then all the response parameters are added to the `redirect_uri` as a URI fragment as shown below.

```
https://app.example.com/redirect_uri#token=XXXXX&id_token=YYYYYY
```

- In addition to the query and fragment, the OAuth 2.0 Form Post Response Mode (https://openid.net/specs/oauth-v2-form-post-response-mode-1_0.html) specification defines another `response_mode` called form_post, and we'll discuss that in chapter 6.
- The `response_type` and `response_mode` are bit related to each other. If you do not specify a `response_mode` parameter in the authentication request, then the default `response_mode` associated with the corresponding `response_type` gets applied automatically. If the `response_type` is `id_token` or `id_token` token (implicit flow) for example, then the corresponding default `response_mode` is fragment (table 3.2). That means, when you use implicit grant flow, the OpenID provider sends back the response parameters as an URI fragment. query string.

Table 3.2: The default response_mode values for the corresponding response_type

The of response_type parameter	The default value of response_mode
id_token token (implicit flow)	fragment
Id_token (implicit flow)	fragment
code (authorization code flow, section 3.9)	query
token (OAuth 2.0 implicit grant type)	fragment

The `response_mode` is an optional parameter in the authentication request, and is originally defined in the OAuth 2.0 Multiple Response Type Encoding Practices specification (https://openid.net/specs/oauth-v2-multiple-response-types-1_0.html), which is developed by the OpenID Foundation (not by the OAuth IETF working group).

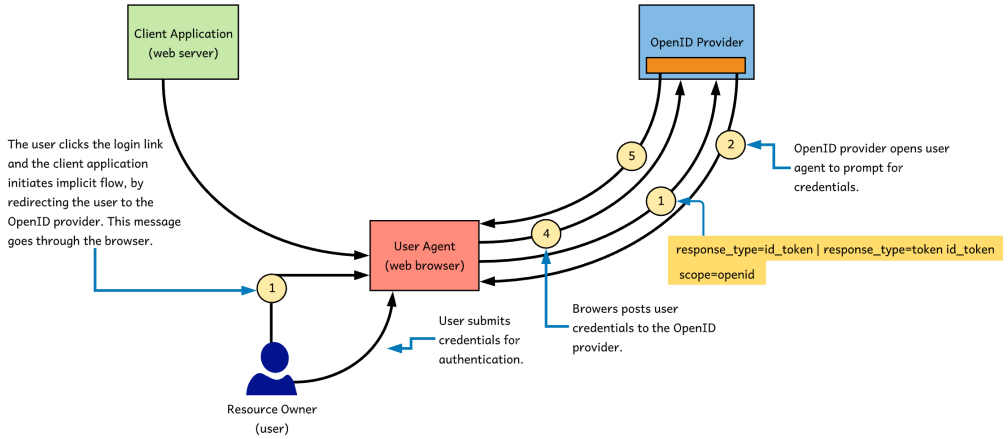

Figure 3.3: The client application uses implicit authentication flow to communicate with the OpenID provider to authenticate the user. This is duplicating the figure 3.1 for readability purpose.

- ***state***: The value of `state` parameter is just a string, which is added to the authentication request by the client application and the OpenID provider must return back the same value (unchanged) in the response (step-5) in figure 3.3. In section 3.5 we discuss the use cases of the `state` parameter and also in chapter 13 we discuss how to use the `state` parameter to mitigate some possible security threats.
- The `state` is an optional, however, a recommended parameter in the authentication request, and is originally defined in the OAuth 2.0 specification.
- ***nonce:*** The value of `nonce` parameter carries a unique value added to the OpenID Connect authentication request by the client application. The OpenID provider must include the value of `nonce` from the authentication request to the ID token it builds. The section 3.7 explains how to generate a nonce to be unique and nonguessable.
- The `nonce` is an optional parameter introduced by the OpenID Connect specification to mitigate replay attacks and in chapter 13 we discuss `nonce` in detail.

In addition to the authentication request parameters we discussed in the above list, there are few more optional ones: ***display***, ***prompt***, ***max_age***, ***ui_locales***, ***id_token_hint***, and ***acr_values***. We'll discuss them in detail in chapter 6.

THE OPENID PROVIDER VALIDATES THE AUTHENTICATION REQUEST AND REDIRECTS THE USER BACK TO THE BROWSER FOR AUTHENTICATION

Once the OpenID provider validates the authentication request from the client application, it checks whether the user has a valid login session under the OpenID provider's domain. Here the domain is the HTTP domain name that you use to access the OpenID provider using a web browser. If the user has logged into the OpenID provider already from the same web browser, then there exists a valid login session, unless its expired.

If the user does not have a valid login session, then the OpenID provider will challenge the user to authenticate (step 2 in figure 3.3); and also will get user's consent to share the requested claims with the client application. In step 3 of figure 3.3 user types the login credentials and in step 4 in figure 3.3, the browser posts the credentials to the OpenID provider. The steps 2, 3 and 4 are out side the scope of the OpenID Connect specification and up to the OpenID providers to implement in the way they prefer. Figure 3.4 shows a sample login page, Google OpenID provider pops up during the login flow.

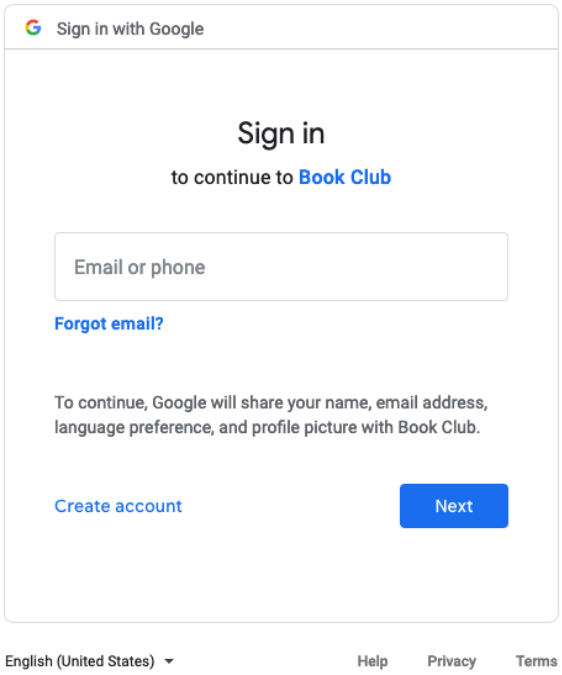

Figure 3.4: A sample login screen for user authentication from the Google OpenID provider.

THE OPENID PROVIDER RETURNS BACK THE REQUESTED TOKENS TO THE CLIENT APPLICATION

In step 5 of figure 3.5, the OpenID provider returns back the requested tokens to the client application. If the client application, for example, requested only an ID token in step 1, by having `id_token` as the value of the `response_type` parameter in the authentication request, then the OpenID provider only returns back an ID token as shown below.

```
https://app.example.com/redirect_uri#id_token=YYYYYYYY&state=Xd2u73hgj59435
```

If the value of the `response_type` parameter was `id_token token`, then the OpenID provider will return back both the ID token and the access token as shown below.

```
https://app.example.com/redirect_uri#access_token=XXXXXX&token_type=Bearer&expires_in=3600&
       id_token=YYYYYYYY&state=Xd2u73hgj59435
```

In both the above cases the OpenID provider returns the tokens as an URI fragment.That's because, if you do not explicitly mention the `response_mode` parameter in the authentication request, for the implicit flow the default value of the `response_mode` is fragment.

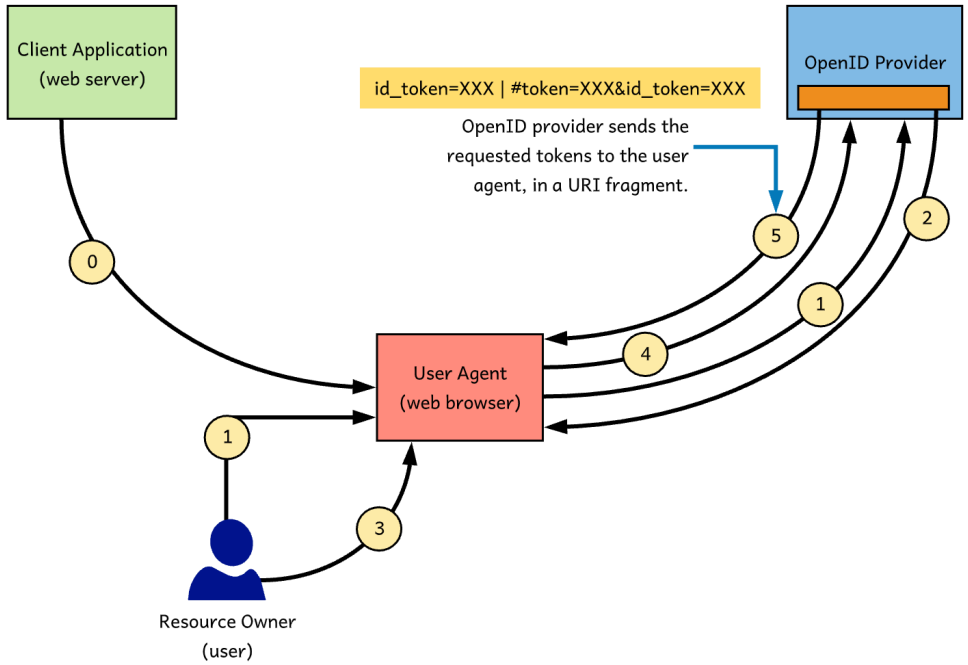

Figure 3.5: In step 5 the OpenID provider returns back the requested tokens to the client application.

Let's go through the parameters in the URI fragments added to the authentication response by the OpenID provider (step-5 in figure 3.5). The definition of these parameters are consistent across all three authentication flows the OpenID Connect defines, however, the values may change.

- ***access_token***: The value of the `access_token` parameter carries the OAuth 2.0 access token. The OpenID provider adds an `access_token` to the response only if the `response_type` is `id_token token`, under the implicit flow. A client application can use this access token to securely access OpenID provider's ***userinfo*** endpoint to retrieve claims with respect to the logged in user or access a business API. In chapter 5, we discuss userinfo endpoint in detail. In case the client application does not need to access any OAuth 2.0 secured APIs, it should use just `id_token` for the `response_type`. So, there won't be any `access_token` in the authentication response.

- As you already learned in chapter 2, what you do with an `access_token` is defined by the `scope`. If you had `email` for the `scope` parameter in the authentication request, for example, then you can use the corresponding `access_token` to access the OpenID provider's ***userinfo*** endpoint to retrieve logged in user's `email` and `email_verified` claims. If you had `address` for the `scope`, then you can retrieve the `address` claim of the logged in user from the ***userinfo*** endpoint.
- The OpenID provider (or the authorization server under the OAuth 2.0 terminology) does not necessarily need to respect the scope value in the authentication request all the time. Based on the consent provided by the user and other policies, the OpenID provider can decide which scope out of all the scopes in the authentication request it wants to respect. So, a client application should not expect all the time to get an access token, which is bound to the quested scope values.
- If the scope of the access token in the response is different from the requested scope, the OpenID provider must include the corresponding scope value in the response, otherwise the client application can safely assume the token is issued for the requested scope values.
- ***id_token***: The value of the `id_token` parameter carries the OpenID Connect ID token, which is a JWT. This is a required parameter and we discuss ID token in detail in chapter 5.
- ***token_type***: The value of the `token_type` parameter carries the type of the OAuth 2.0 access token. This is a required parameter and we discussed this in detail in chapter 2.
- ***expires_in***: The value of the `expires_in` parameter carries the validity of the OAuth 2.0 access token in seconds calculated from the time it is issued. This is an optional, but a recommended parameter and we discussed this in detail in chapter 2.
- ***state***: The value of the `state` parameter copies the value of the `state` parameter from the authentication request. The value of the `state` parameter in response must be exactly the same found in the request. In section 3.6 we discuss how to use the `state` parameter. This is a required parameter only if the authentication request carries a `state` parameter.
- ***scope***: If the scope of the access token in the response is different from the requested scope, the OpenID provider must include the corresponding scope value in the response, otherwise the client application can safely assume the token is issued for the requested scope values.

One important thing you might have already noticed in the authentication response from the OpenID provider is, there is no `refresh_token`. A `refresh_token` is defined in the OAuth 2.0 specification and is used by the client applications to refresh (extend the token expiration) the `access_token` and the `id_token`. However, the implicit flow in OpenID Connect does not return back an `refresh_token`.

If you don't have a `refresh_token`, the client application won't be able to renew the `access_token` (or the ID token) it got from the OpenID provider, and in that case the client application has to initiate a new authentication request to get a new `access_token` and a ID token. This is one of the reason (not the only reason, we discuss more in section 3.11)

people prefer to use the authorization code flow over implicit flow. We discuss authorization code flow in detail in section 3.9 and refresh tokens in detail in chapter 6.

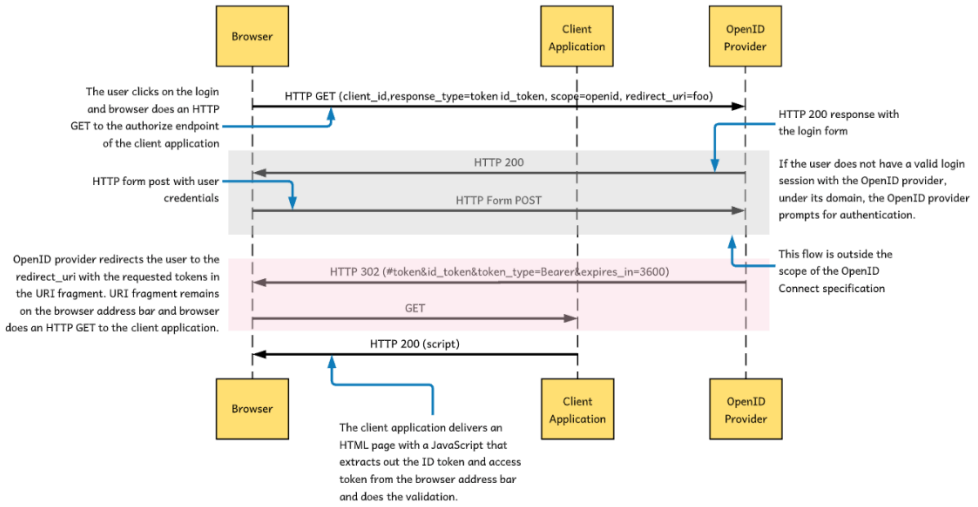

Figure 3.6: The client application uses implicit authentication flow to communicate with the OpenID provider to authenticate the user.

Once the client application gets the tokens in the authentication response, it can use a JavaScript to extract out the `access_token` and the `id_token` from the URL fragment. In practice, what happens is, the OpenID provider does an HTTP redirect (with 302 status code) to the `redirect_uri` corresponding to the client application and the client application delivers an HTML page with a JavaScript to the browser, which extracts out the ID token and the access token from the URI fragment and does the validation (see figure 3.6).

The following code listing shows a JavaScript code segment extracted out from the OpenID Connect Implicit Client Implementer's Guide 1.0 (https://openid.net/specs/openid-connect-implicit-1_0.html) that extracts out the URI fragment from the browser location bar and posts to a backend endpoint for validation.

Listing 3.2: A JavaScript code that validates the tokens in the URI fragment

```
<script type = "text/javascript" >

// First, parse the query string
var params = {},
    postBody = location.hash.substring(1),
    regex = /([^&=]+)=([^&]*)/g,
    m;
while (m = regex.exec(postBody)) {
    params[decodeURIComponent(m[1])] = decodeURIComponent(m[2]);
}

// And send the token over to the server
var req = new XMLHttpRequest();
// using POST so query isn't logged
req.open('POST', 'https://' + window.location.host + '/catch_response', true);
req.setRequestHeader('Content-Type','application/x-www-form-urlencoded');

req.onreadystatechange = function(e) {
    if (req.readyState == 4) {
        if (req.status == 200) {
            // If the response from the POST is 200 OK, perform a redirect
            window.location = 'https://' +
                window.location.host + '/redirect_after_login'
        }
        // if the OAuth response is invalid, generate an error message
        else if (req.status == 400) {
            alert('There was an error processing the token')
        } else {
            alert('Something other than 200 was returned')
        }
    }
};
req.send(postBody)
</script>
```

3.4 Why does one client application need to have multiple redirect_uris?

The `redirect_uri` is a required parameter in the authentication request both under the implicit flow we discussed in section 3.3 as well as under the authorization code flow we are going to discuss in the section 3.9. As you already learnt in section 3.3, the OpenID provider validates the `redirect_uri` in the authentication request against the one which is already registered at the OpenID provider (during client application registration flow).

In theory, the OpenID provider is expected to do one-to-one match between these two `redirect_uri`s; however, in practice, sometimes we see one client application register multiple `redirect_uri`s. This is a common pattern in the enterprise use cases where the organizations have multiple applications managed by the same team. In such cases, they don't want to duplicate the application configuration at the OpenID provider. Or in other words they do not want to register each and every client application at the OpenID provider; rather, they want to reuse one `client_id`, with multiple `redirect_uri`s by different applications (see figure 3.7).

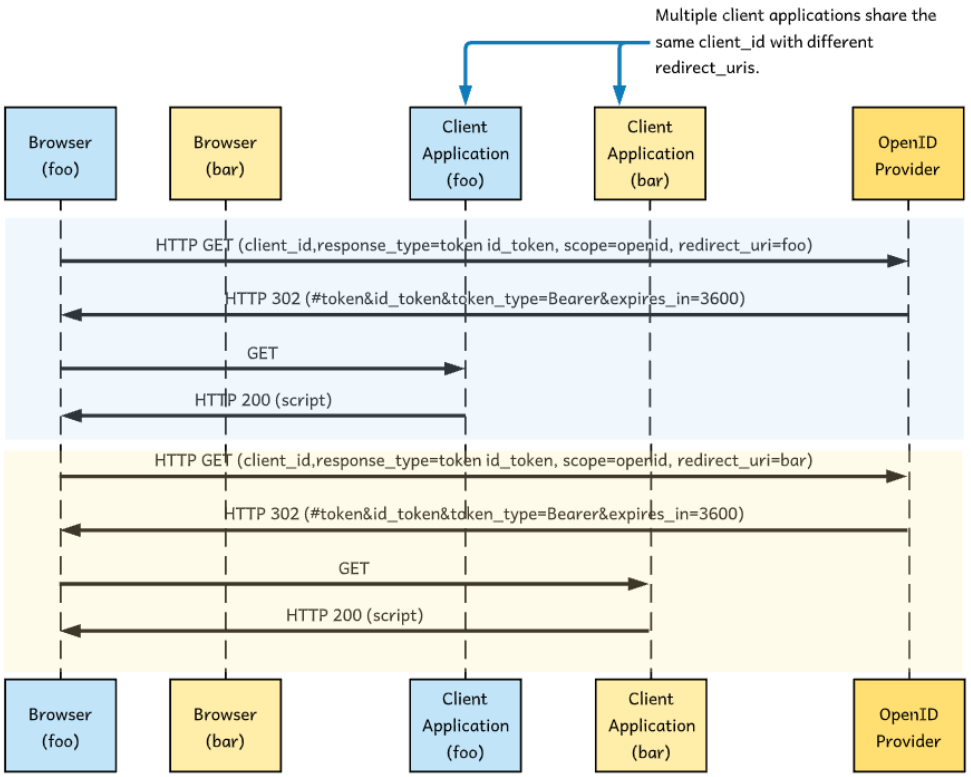

Figure 3.7: Multiple client applications reuse the same client_id but with different redirect_uris.

If you are to follow this pattern, you need to do this cautiously by knowing the drawbacks of this approach, as listed below.

- Since you are using a single `client_id` to identify multiple applications, OpenID provider won't be able to recognize authentication requests generated by different applications independently. Even though still it's possible to differentiate authentication requests from each other by looking at the `redirect_uri`, in practice, most of the OpenID providers are not built in that way.
- Most of the OpenID providers make the `access_token` unique by the `client_id`, user, associated scopes and the status of the token. The status of a token can be ACTIVE, EXPIRED, REVOKED, LOCKED and so on. None of these are defined in the OpenID Connect or OAuth 2.0 specifications; rather implemented by different OpenID providers in the way they want. For a given `client_id`, user, set of scopes and status, OpenID provider can find one `access_token`.

- Then again, if the same user tries to login to the same client application several times even before the first issued token is expired, the number of tokens the OpenID provider has to maintain could explode, if the OpenID provider generates a new token for each login of the user. To fix this token explosion problem, some OpenID providers return the same `access_token` (not the ID token) back to the client application for subsequent login requests from the same client application for the same user for the same set of scopes, if the original token is not expired.
- Under the context of this topic, the above solution to the token explosion problem will result in sharing the same `access_token` with multiple applications, because they do share the same `client_id`. One workaround for this is, use one additional scope value to represent the client application – and it has to be unique for a given application. This is a special scope value, which acts a signal to the OpenID provider, rather a scope that requires user's consent. Since the requested scopes are different by the application, even though they share the same `client_id`, still OpenID provider will generate different tokens by application.
- Even if you register multiple `redirect_uri`s at the OpenID provider, still the OpenID provider does one-to-one validation of the `redirect_uri` in the authentication request against each of the registered `redirect_uri`s to see whether there is a match.
- If the OpenID provider provides an option to validate the `redirect_uri` in the authentication request with a regular expression pattern instead of doing one-to-one match between the `redirect_uri` registered at the OpenID provider and the `redirect_uri` in the authentication request, that could help an attacker to plant an attack by changing the `redirect_uri` in the authentication request by adding some parameters, that will still pass the regular expression pattern. We'll discuss such possible attacks in chapter 13.
- So, if you are using a regular expression pattern for `redirect_uri` validation, you need to make sure that it's tested properly only to include what you want to have. However, as discussed in chapter 2, with OAuth 2.1, the authorization server (or the OpenID provider) must make sure that the `redirect_uri in` the authorization request exactly matches with the `redirect_uri` registered at the authorization server.

3.5 Using the state parameter

As discussed in the section 3.3 the OpenID provider is obliged to return the value of the `state` parameter from the authentication request, in the authentication response back to the client application. From the client application's point of view, this provides a way to correlate an authentication response from the OpenID provider with the authentication request it generates (figure 3.8).

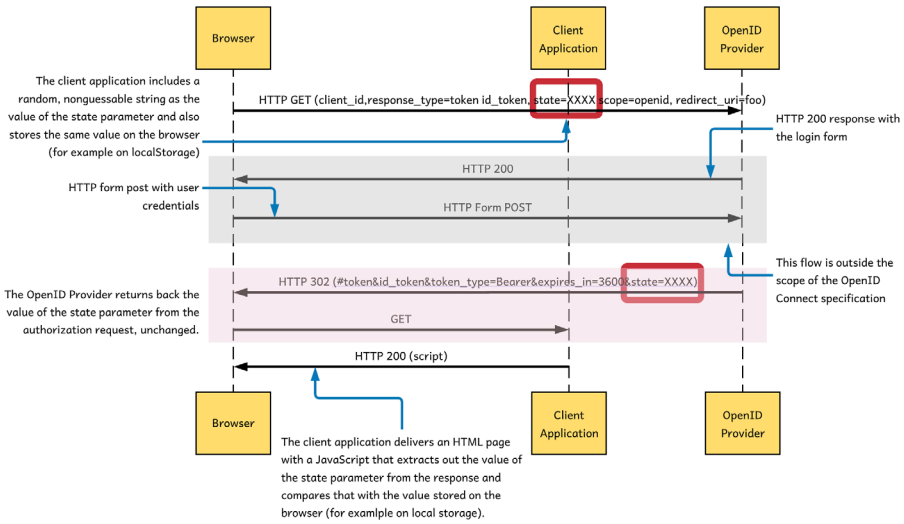

Figure 3.8: The client application adds a random, nonguessable string as the value of the state parameter to the authentication request, and the OpenID provider returns the same value in the authentication response, back to the client application.

Following lists out some use cases of the `state` parameter.

- In an ecommerce application, for example, a user can add certain items to the shopping cart and at the point they decide to checkout, the login with OpenID Connect will kick in and will redirect the user to the OpenID provider. However, when the user returns back from the OpenID provider to the client application, the user expects the same shopping cart with all the items they picked, before being redirected to the OpenID provider.
- In a typical web application, this is handled by maintaining a session with the backend web server, and correlated with the browser via a session cookie. In a SPA, you can use the HTML5 session storage of the browser to store the shopping cart items against a key, which is a randomly generated identifier. The value of the key is non-guessable and non-static, and generated once for each browser session.
- This key can go as the value of the `state` parameter in the authentication request; and when the user returns back from the OpenID provider, the client application can find the `state` value from the authentication response and use that as a key to load the saved shopping cart from the browser session storage.
- In section 1.9.4 of chapter 1 we discussed bootstrap trust with external identity providers as a benefit of having a single trusted identity provider for a client application. So, the client application only connects to its own trusted OpenID provider, and then that OpenID provider helps connecting the client application to other external identity providers (figure 3.9). These external identity providers can be OpenID Connect, SAML and so on.

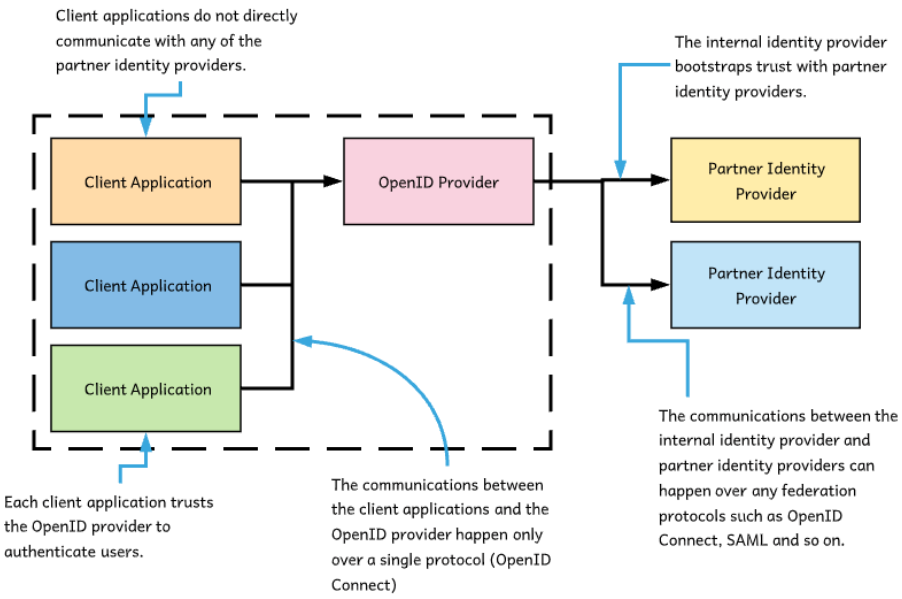

Figure 3.9: The internal identity provider bootstraps trust with partner identity providers, and does claim/protocol transformations, as expected by the client applications connected to it. All the client applications only need to trust the internal identity provider, and should only need to support the federation protocol, the internal identity provider supports.

- The OpenID provider here is responsible for getting a response from the external identity providers and transforming the response in a way the response it (the OpenID provider) generates is understood by the first client application. To do that the OpenID provider has to cache (or store) the initial request it got from the client application and should able to correlate it to the response it gets from the external identity provider.
- Most of the OpenID providers implement this using the `state` parameter in the authentication request. Before the OpenID provider redirects the user to an external identity provider, it generates a random string as the correlation handle, and stores the authentication request from the client application against it. The OpenID provider adds the correlation handle as the value of the `state` parameter in the authentication request it generates, and when it receives a response from an external identity provider, the OpenID provider expects the same correlation handle to be in the `state` parameter of the response; and that helps to retrieve the initial authentication request from the cache.

- Finally, and most importantly, one of the primary use cases of the `state` parameter is security. The `state` parameter helps protect a client application from a cross-site request forgery (CSRF) attack, which we will discuss in detail in chapter 13.

3.6 URI fragment vs. query string

In the section 3.3.1 you learnt that the `response_mode` parameter in the authentication request, dictates the structure or the format of the response a client application gets from an OpenID provider. If you set the value of the `response_mode` parameter to ***query***, for example, then all the parameters in the response are encoded as a query string added to the `redirect_uri` as shown below.

```
https://app.example.com/redirect_uri?token=XXXXX&id_token=YYYYYY
```

If you set the value of the `response_mode` parameter to ***fragment***, then all the response parameters are added to the `redirect_uri` as a URI fragment as shown below.

```
https://app.example.com/redirect_uri
       #access_token=XXXXXX&token_type=Bearer&expires_in=3600&id_token=YYYYYYYY&state=Xd2u7
       3hgj59435
```

When the OpenID provider redirects the user back to the client application, it sets the HTTP response status code to 302 and sets the `Location` header to the `redirect_uri` either with a query string or with an URI fragment. Then the browser after receiving 302 from the OpenID provider, extracts out the URL from the `Location` header and does an HTTP GET. Following lists out the key differences between an URI fragment and a query string.

- Any parameter in a URI fragment never leaves the browser. When the browser does an HTTP GET to the URL comes from the Location header of the 302 response from the OpenID provider, the HTTP GET request goes to the backend web server, but still the URI fragment remains on the browser address bar. However, a query string attached to a URL, goes to the backend web server.
- As defined in the RFC 2616 (https://tools.ietf.org/html/rfc2616), the HTTP `Referer` header does not carry anything from the URI fragment, but everything in the query string is included.
- Usually, when you click on a link on a web page, the URL of the current web page goes as the `Referer` header of the HTTP request to the new web page, unless you have setup a policy to not to include the `Referer` header. Having certain information in the `Referer` header could lead to possible security issues and in chapter 10 we discuss them in detail and how to mitigate those.

3.7 Generating a random, unguessable nonce

As discussed in the section 3.3.1 value of the `nonce` parameter carries a unique value added to the OpenID Connect authentication request by the client application. In this section you'll learn how to generate a random, nonguessable nonce value in Java; and you'll be able to find similar constructs to generate random numbers in other programming languages as well.

When picking an algorithm to generate a random number, there are few properties you need to worry about. You do not need to implement them by your own. Most of the random number generators available in different programming languages support these properties.

- A random number should be **statistically independent**. That means, the random number generator must not rely on previously generated ones to generate new random numbers.
- A random number should be **unpredictable**. No one should be able to guess a random number by looking at a random number already generated by the corresponding random number generator.
- The generated random numbers should be **uniformly distributed**. The probability of generating a given random number should be equal among all the possible random numbers, the random number generator generates.

The following code listing shows how to use the `java.secure.SecureRandom` Java class to generate a random number. Neil Madden, the author of the book API Security in Action (Manning, 2020) suggests this approach in his blog (https://neilmadden.blog/2018/08/30/moving-away-from-uuids/).

Listing 3.3: Generating a random, nonguessable nonce in Java

```
import java.security.SecureRandom;
import java.util.Base64;

public class SecureRandomString {

private static final SecureRandom random = new SecureRandom();
private static final Base64.Encoder encoder=Base64.getUrlEncoder().withoutPadding();

public static String generateNonce() {
byte[] buffer = new byte[20];
             random.nextBytes(buffer);
             return encoder.encodeToString(buffer);
           }
       }
```

3.8 Implementing implicit flow using Google as the OpenID provider

In this section we discuss how to implement implicit authentication flow by using Google as an OpenID provider. Here we are not going to build a web application from ground-up; rather, we use a row URL constructed from all the necessary parameters and place it on the browser address bar to demostrate the authentication request and authentication response we discussed in section 3.3 under the implicit flow.

3.8.1 Setting up Google as an OpenID provider

To set up Google as an OpenID provider, please check https://github.com/openidconnect-in-action/samples/blob/master/IDPs.md. Once you are done with that, you get a `client_id` and a `client_secret` for your client application. However, since we are using implicit authentication flow here, we only need `client_id`. Also, while registering your client

application with Google, you also need to provide one or more `redirect_uri`s and lets assume you have https://localhost:3000/redirect_uri as the `redirect_uri`. The following code snippet lists all the parameters we need to know to build a client application against the Google OpenID provider following the implicit flow.

Listing 3.4: Parameters required to communicate with Google OpenID provider

```
client_id: 450443992251-7gft9u.apps.googleusercontent.com
redirect_uri: https://localhost:3000/redirect_uri
Google authorization endpoint: https://accounts.google.com/o/oauth2/v2/auth
```

3.8.2 Constructing the authentication request

In this section we'll construct an OpenID Connect authentication request with the parameters from the listing 3.4 and some additional parameters such as `scope`, `state`, `response_type`, and `nonce`. The following listing shows the complete authentication request; you may replace the value of `client_id` with what you got from Google after registering the client application. Also, we use two random string values for `state` and `nonce` parameters, and we expect the OpenID response to carry the same `state` value. In section 3.7 we discussed how to generate random, nonguessable value for the `nonce` parameter.

Listing 3.5: Authentication that redirects the user to the Google OpenID provider

```
https://accounts.google.com/o/oauth2/v2/auth?
client_id=450443992251-7gft9u.apps.googleusercontent.com&
redirect_uri=https://localhost:3000/redirect_uri&
scope=openid profile&
response_type=id_token token&
state=caf7871khs872&
nonce=89hj37b3gd3
```

Once you copy and paste the above request into your browser location bar, it will take you to Google (the OpenID provider) for authentication. If you are not logged in already, you may see a screen similar to figure 3.5. Also, the same screen will display the name of the client application you used during the application registration process, and the set of attributes Google about to share with the client application. We used `profile` as the `scope` value (in addition to `openid`) in the authentication request, and the list of attributes (name, email address and so on) shown in figure 3.10 are related to that. You'll learn more about requesting user attributes using scopes in chapter 5.

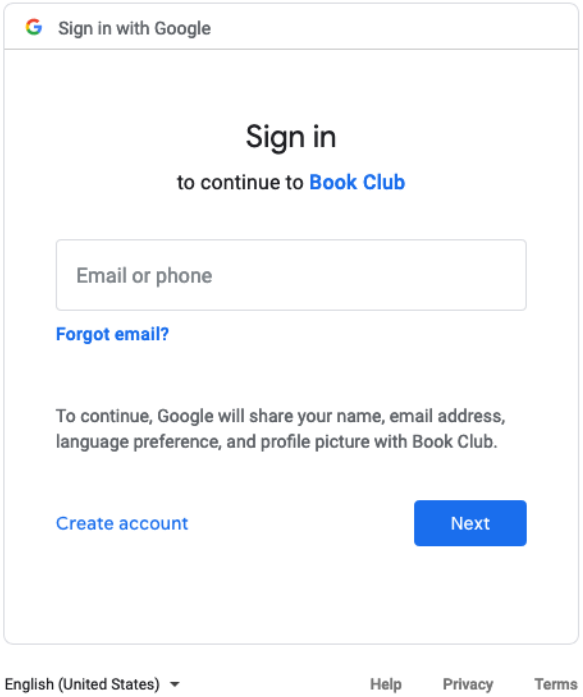

Figure 3.10: Google login screen for user authentication. The name of the client application you used during application registration process is displayed on the screen (Book Club), along with the user attributes Google is going to share with the client application.

Once you complete the authentication flow at the Google OpenID provider by entering your own credentials, it will redirect you back to the client application (to the `redirect_uri` we provided). However, since we do not have any application running on that address, the response from Google OpenID provider will remain on the browser location bar, as shown in the following code (listing 3.6). In practice, you will get a lengthy string for the values of the `access_token` and `id_token` parameters, which we have replaced in the listing 3.6 with the text `<ACCESS_TOKEN>` and `<ID_TOKEN>` respectively. In the response from Google, we got both the `access_token` and the `id_token`; as you might have rightly guessed already, that is because we used `id_token token` as the `response_type` in the authentication request (listing 3.5). In case we have used just `id_token` as the `response_type`, you won't see the `access_token` in the response.

Listing 3.6: Authentication response from the Google OpenID provider

```
https://localhost:3000/redirect_uri#state=caf7871khs872
      &access_token=<ACCESS_TOKEN>
      &token_type=Bearer
      &expires_in=3599
      &scope=profile%20openid%20https://www.googleapis.com/auth/userinfo.profile
      &id_token=<ID_TOKEN> &authuser=0&prompt=consent
```

3.8.3 An overview of the ID token returned back from the Google OpenID provider

In section 3.8.2 we got an authentication response from the Google OpenID provider, which includes an ID token. In this section we'll delve deep into the attributes that you find in the ID token (listing 3.6). As you learnt in chapter 1, the ID token is a JSON Web Token (JWT). In chapter 4, we discuss JWT in detail.

The ID token you got from Google has three parts in it, where they are separated by each other with a period (.). If you have three parts in a JWT, that is a JSON Web Signature (JWS) and if you have five parts in a JWT, that is a JSON Web Encryption (JWE). In chapter 4, you learnt about both JWS and JWE in detail.

To decode the JWT you got in the response from Google, in listing 3.6, let's use the https://jwt.io. Just copy the value of `id_token` parameter in the Encoded text area of the web site (jwt.io), and you will get the values decoded. Then again, you need to treat your ID tokens as secrets and never use public sites like jwt.io to decode any production ID tokens. [4]Once you decode the ID token, you will see the decoded payload of the JWT as shown in the following listing.

Listing 3.7: The decoded payload of the ID token returned from Google

```
{
  "iss": "https://accounts.google.com",
  "azp": "450443992251-9l17d82cli8npa9cdrvcp1g9m17gft9u.apps.googleusercontent.com",
  "aud": "450443992251-9l17d82cli8npa9cdrvcp1g9m17gft9u.apps.googleusercontent.com",
  "sub": "104063262378861625904",
  "at_hash": "Krnm3SB1v_UR00j50VzLoQ",
  "nonce": "89hj37b3gd3",
  "name": "Prabath Siriwardena",
  "picture": "https://lh3.googleusercontent.com/a-
      /AOh14GiriDTmbf8tcSKzMkFYvYwYuBMUmGFdtEBqpvRGOA=s96-c",
  "given_name": "Prabath",
  "family_name": "Siriwardena",
  "locale": "en",
  "iat": 1601313517,
  "exp": 1601317117,
  "jti": "4fea08bd4386e45d6f8b869520ace3f7a4f80bde"
}
```

The JWT payload in listing 3.7 has two types of claims: the claims related to the end user and the claims related to token validation. The `sub`, `name`, `picture`, `given_name`, `family_name`, and `locale` are related to the end user, and all those claims related to the

4 You can follow the approach this blog (https://prefetch.net/blog/2020/07/14/decoding-json-web-tokens-jwts-from-the-linux-command-line/) suggests to decode a JWT in the command line. That's a much safer way to decode a JWT, than relying on online tools.

user are standard claims defined in the OpenID specification. In chapter 5 we'll go through these claims in detail.

The rest of the claims in listing 3.7 are defined in two specifications. Some are defined in the JWT specification (`jti`, `iat`, `exp`, `aud`, `iss`) and some are defined in the OpenID Connect specification (`nonce`, `at_hash`, `azp`). Chapter 4 covers in detail all the claims defined by the JWT specification, and in the section 3.7.4, we'll explain `nonce`, `at_hash` and `azp`. Even though `iat`, `exp`, `aud` and `iss` claims are defined in the JWT specification, the OpenID Connect specification makes all of them mandatory in an ID token.

3.8.4 ID token validation rules

You already know that an ID token is a JWT. So, the first step in validating an ID token is to make sure, it is a valid JWT. We discuss in detail how to validate a JWT in chapter 4. This section explains how a client application uses some of the attributes OpenID Connect specification introduced into the ID token during the validation process. Once the client application identified the ID token as a valid JWT it needs perform further validation as listed in the following.

- If the authentication request included a `nonce` parameter, the same must be returned to the client application as a claim in the ID token; and also the client application must make sure it has not seen the same `nonce` value in an ID token before. This check prevents a replay attack of an ID token, which we discuss in detail in chapter 13.
- The value of the `at_hash` parameter in the ID token is a base64url-encoded hash of a part of the access token (not the hash of the complete token). The hashing algorithm the OpenID provider uses to hash the token is included in the JWT header under the `alg` parameter. If an `access_token` is returned along with ID token in the authentication response from the OpenID provider, this parameter must be present in the ID token and must be valid. This helps preventing some of the security issues found in the OAuth 2.0 implicit grant type and chapter 13 discusses that in detail.
- The `azp` parameter (or the authorized party) in the ID token is an optional parameter; but if the `client_id` is not present in the `aud` parameter of the ID token the `azp` parameter must be included in the ID token and its value must be the `client_id`.

There are few more claims (`auth_time`, `act` and `amr`) OpenID Connect specification introduced with respect to token validation and we'll discuss them in detail in chapter 6. Then again most of the time as an application developer, you will not write code to do these validations yourself, rather will use some OpenID Connect library.

Security issues with the implicit flow and how to mitigate those

We discussed OAuth 2.0 implicit grant type in chapter 2. The IETF OAuth 2.0 working group recommends not using the implicit grant type due to couple of security issues explained in the OAuth 2.0 Security Best Current Practice document (https://tools.ietf.org/html/draft-ietf-oauth-security-topics-15), mostly the access token replay and access token leakage attacks. We discuss these two attacks in chapter 13. However, the implicit flow in OpenID Connect is not exactly the implicit grant type in OAuth 2.0; and OpenID Connect adds protection to prevent the implicit flow from both the access token replay and access token leakage attacks.

The `at_hash` parameter we discussed in section 3.7.4 binds the access token issued along with the ID token, to the ID token. Since the ID token itself has replay protection with the `nonce` parameter, binding the access token to the ID token, also protects the access token being replayed. We have dedicated chapter 13 to discuss security issues related to OpenID Connect and OAuth 2.0; so, we'll differ a detailed discussion on this to chapter 13.

3.9 How does authorization code flow work?

In this section we'll take you through how the authorization code flow works with a SPA in detail. There are many similarities in the parameters passed in the authentication request and authentication response, both under the implicit flow we discussed in section 3.3 and the authorization code flow. So, we assume you have gone through the section 3.3 already. The sequence of events or steps that happens during this flow, as well as the messages being passed in each step, are clearly defined in the OpenID Connect specification.

3.9.1 The flow of events in the authorization code authentication flow

Figure 3.6 lists the sequence of events happens between the OpenID provider, the client application, and the user. The client application in figure 3.6 can be any type of an application, but here our discussion mostly focuses on a SPA. Over the time, the authorization code flow has become more popular for implementing login with OpenID Connect for SPAs. In the following sections we discuss what happens in each step in the figure 3.11 in detail.

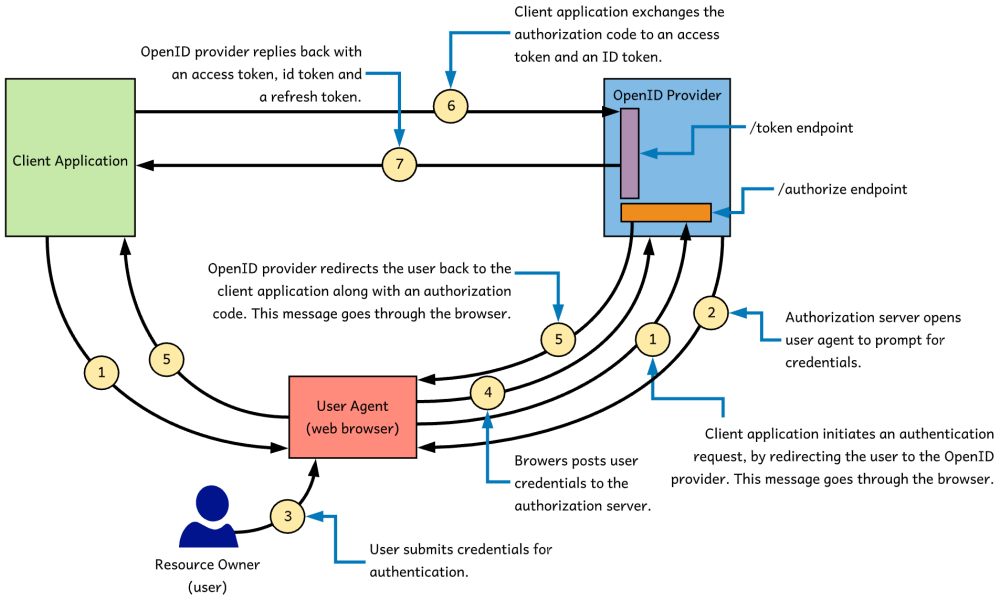

Figure 3.11: The client application uses authorization code authentication flow to communicate with the OpenID provider to authenticate the user.

THE CLIENT APPLICATION INITIATES A LOGIN REQUEST VIA THE BROWSER (STEP 1)

In the step 1 of figure 3.11, the client application initiates a login request via the browser. In case of a SPA, we can expect the user clicks on a login link on the web page of the client application, and browser does an HTTP GET to the ***authorize*** endpoint of the OpenID provider. Listing 3.3 shows an example of an authentication request under the authorization code authentication flow.

Listing 3.3: Authentication request generated by the client application (authorization code flow)

```
https://accounts.google.com/o/oauth2/v2/auth?
      response_type=code&
      client_id=424911365001.apps.googleusercontent.com&
      scope=openid email&
      redirect_uri=https%3A//app.example.com/redirect_uri&
      state=Xd2u73hgj59435&
      login_hint=jdoe@example.com&
      nonce=0394852-3190485-2490358
```

In section 3.3 we discussed the usage of all the parameters in the listing 3.3, so we do not intend to duplicate the same here. However, a few parameters need attention:

- ***response_type***: The value of the `response_type` parameter in the authentication request defines which tokens the authorization endpoint of the OpenID provider should return back to the client application.

- In the authorization code flow there is only one possible value: ***code***. The client application expects that authorization endpoint of the OpenID provider to return a code in the authentication response. This is the key parameter in the authentication request that differentiates an authorization code flow from the implicit flow.
- ***response_mode***: The value of the `response_mode` parameter in the authentication request defines how the client application expects the response from the OpenID provider. For the authorization code flow, the default value of `response_mode` parameter is `query`. So, the client application expects the OpenID provider to return the code and the corresponding parameters in query string to the `redirect_uri`.

THE OPENID PROVIDER VALIDATES THE AUTHENTICATION REQUEST AND REDIRECTS THE USER BACK TO THE BROWSER FOR AUTHENTICATION (STEP 2)

Once the OpenID provider validates the authentication request from the client application, it checks whether the user has a valid login session under the OpenID provider's domain. If the user has logged into the OpenID provider already from the same web browser, then there exists a valid login session, unless its expired.

If the user does not have a valid login session, then the OpenID provider will challenge the user to authenticate (step 2 in figure 3.12); and also will get user's consent to share the requested claims with the client application. In step 3 of figure 3.12 user types the login credentials and in step 4 in figure 3.12, the browser posts the credentials to the OpenID provider. The steps 2, 3 and 4 are out side the scope of the OpenID Connect specification and up to the OpenID providers to implement in the way they prefer.

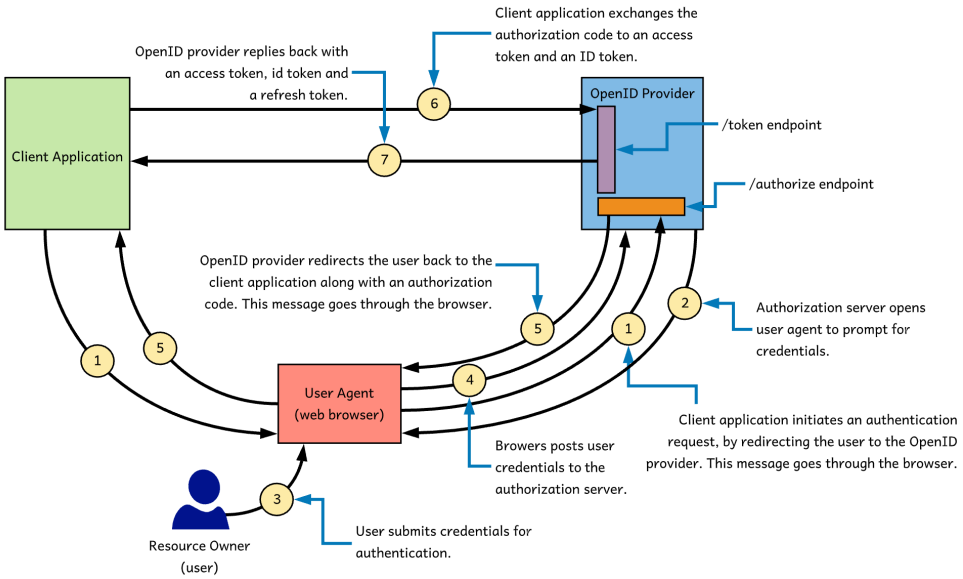

Figure 3.12: The client application uses authorization code authentication flow to communicate with the OpenID provider to authenticate the user.

THE OPENID PROVIDER RETURNS BACK THE AUTHORIZATION CODE TO THE CLIENT APPLICATION (STEP 5)

In step 5 of figure 3.12, the OpenID provider returns the authorization `code` along with the `state` parameter to the client application in a query string to the `redirect_uri` (as shown in the following line of code). Once the the client application receives the request, it needs to make sure that the value of the state parameter in the response exactly matches the value of the state parameter in the authentication request.

```
https://app.example.com/redirect_uri?code=YDed2u73hXcr783d&state=Xd2u73hgj59435
```

THE CLIENT APPLICATION EXCHANGES THE AUTHORIZATION CODE TO AN ID TOKEN AND AN ACCESS TOKEN (STEP 6)

Unlike in the implicit flow, in the authorization code flow, the client application does not get the ID token or the access token in the authentication response. To get an ID token and an access token, the client application has to talk to the token endpoint of the OpenID providers, as shown in the step 6 of figure 3.12. Following listing shows an example request to the token endpoint of the OpenID provider, which carries the authorization code from the authentication response. The token request defined in the authorization code authentication flow in OpenID Connect is identical to the token request defined under the authorization code grant type in OAuth 2.0 specification, which we discussed in the chapter 2.

Listing 3.4: Request to the token endpoint of the OpenID provider (authorization code flow)

```
POST /token
HTTP/1.1
Host: oauth2.googleapis.com
Content-Type: application/x-www-form-urlencoded

code=YDed2u73hXcr783d&
client_id=your_client_id&
redirect_uri=https%3A//oauth2.example.com/code&
grant_type=authorization_code
```

One key thing to notice here is, in the token request in listing 3.4, the client application does not authenticate to the OpenID provider. So, we only use the client_id in the request and the client application does not need to have a client_secret or any other authentication mechanism. As you already learnt in chapter 2, an SPA is called a public client under the OAuth 2.0 specification and a public client does not have the capability protect any secrets.

Since a SPA runs the browser, it can't hide any secrets from the end user. Anything that you hide on the browser is visible to the end user. So, no point of having any credentials for an SPA. In chapter 6, we'll discuss how to use the authorization code flow with a traditional (server-side) web application and there web application will use client_id and client_secret to authenticate to the token endpoint of the OpenID provider.

THE OPENID PROVIDER RETURNS BACK AN ID TOKEN AND ACCESS TOKEN TO THE CLIENT APPLICATION (STEP 7)

In step 7 of figure 3.8 the OpenID provider returns back an ID token and an access token to the client application, as shown in the following listing.

Listing 3.5: The response from the token endpoint of the OpenID provider

```
{
  "access_token": "1/fFAGRNJru1FTz70BzhT3Zg",
  "expires_in": 3920,
  "token_type": "Bearer",
  "id_token": "<ID_TOKEN>"
  "refresh_token": "<ACCESS_TOKEN>"
}
```

The only difference you see in the response in listing 3.5 and the response you get from the token endpoint under the OAuth 2.0 authorization grant type (which we discussed in chapter 2), here we have an additional parameter called `id_token`, which carries the ID token. Also, unlike in the implicit flow, in the authorization code flow, the client application also gets a `refresh_token`, which can be used to renew the `access_token` and also the `id_token`. We talk about refreshing tokens in detail in chapter 6.

3.10 Authorization code flow or the implicit flow?

In section 3.3 we discussed implicit flow in detail and in section 3.9 the authorization code flow in detail. In practice, there are SPAs that use implicit flow as well as the authorization code flow. However, most of the new SPAs use authorization code flow, and based on the following points listed out we recommend using authorization code flow over the implicit flow.

- Under implicit flow, the `access_token` and the `id_token` are returned as an URI fragment to the `redirect_uri`. The URI fragment will remain in the browser history, and anyone having access to the browser could observe those. However, from the following JavaScript code running on client application, you still can remove the URI fragment (after reading the tokens) from the browser location bar as well as from the browser history.

```
// remove fragment as much as it can go without adding
// an entry in browser history:
window.location.replace("#");

// slice off the remaining '#' in HTML5:
if (typeof window.history.replaceState == 'function') {
  history.replaceState({}, '', window.location.href.slice(0, -1));
}
```

- Implicit flow does not return back a `refresh_token`, while the authorization code flow does. Under implicit flow if the `access_token` expires, then the client application has to re-initiate the login flow to obtain a new `access_token`.
- This point is not directly related to SPA, however if you implement OpenID Connect login with the implicit flow in a native mobile application or a desktop application, it could possibly be vulnerable for token interception, which we discuss in detail in chapter 13. Under authorization code flow, this attack can be mitigated by implementing Proof Key for Code Exchange (PKCE) OAuth 2.0 profile (https://tools.ietf.org/html/rfc7636) that we discuss in chapter 6 in detail.

- Finally, its better to understand why the implicit flow was added to the OpenID Connect specification. When you use the authorization code flow with a SPA, to exchange the code to an ID token or/and an access token, you need to do a direct HTTP request from a JavaScript running on the browser to the token endpoint of the OpenID provider. Most of the time, the domain of the token endpoint is different from the domain of the client application. So, by default browsers won't permit cross-domain calls. That means a client application running on app.example.com domain, won't be able to do a direct HTTP request from the JavaScript to the op.example.com domain.
- That's was the reason, the OpenID Connect introduced the implicit flow. Unlike in the authorization code, with implicit flow there is no direct HTTP requests between a client application running on the browser and the OpenID provider. However, over the time cross-origin resource sharing (CORS) policies became popular. CORS enables you to do cross domain calls selectively. So, you can use authorization code flow by enabling CORS for the client application's domain to access the token endpoint of the authorization server. That means, there is no reason to use the implicit flow now. In chapter 5, you'll learn more details about CORS.

3.11 Securing a single-page application using OpenID Connect

In this section you'll learn how to create a fully functioning SPA using React and then integrate the SPA with OpenID Connect for login. We assume you have a good knowledge on React; if not, please go through appendix A first, which covers all the React fundamentals you need to know to follow this section.

3.11.1 Building a single-page application with React

In this section we'll build a React application with no security, and make sure its up and running. All the samples we use in this book are available in the GitHub repository: https://github.com/openidconnect-in-action/samples; either you can do a `git clone` or download all the samples as a zip file. The sample we discuss in the section is available under the chapter03/sample01 directory. To build the same React application, run the following command from the sample01 directory. This `npm` command will look at the `package.json` file inside the sample01 directory and download all the dependent node modules and store them under the directory sample01/node_modules. You won't see the node_modules directory in the samples you downloaded from the GitHub repository; it's created only after you run the following command.

```
\> npm install
```

To build the React application, run the following command from sample01 directory on the command console. This command wll create directory called b`uild` and copy all the files that you want to deploy into your production web server.

```
\> npm run build
```

In this example we use a node server as our web server. You can start using the following command run from the sample01 directory.

```
\> npm start
```

The above command starts the node server on localhost port 3000 by default; if you visit http://localhost:3000 link on the web browser, you will see a welcome message. This is the simplest React application you can have; in the next two sections, you'll learn how to secure this application with OpenID Connect.

3.11.2 Setting up an OpenID Provider

To secure the React application we developed in section 3.11.1, we need to have an OpenID provider. In section 3.8, in our discussion of the implicit flow, we used Google as the OpenID provider. However, we cannot use Google as an OpenID provider here, because Google always expects the client application that implements OpenID Connect login with Google, following the authorization code flow, to authenticate to the token endpoint using client_id and client_secret.

As we discussed in section 3.9, a SPA is a public client and public clients cannot securely store secrets. So, there is no point of authenticating a SPA. Here we need to use an OpenID provider that supports the authorization code flow with no client authentication. Here (check https://github.com/openidconnect-in-action/samples/blob/master/IDPs.md), we explain how to setup two open source OpenID providers and to run the examples in this section you can pick one of them. Once you successfully set up your OpenID provider and register your application with the OpenID provider, you need to have following parameters to secure access to the React application with OpenID Connect.

Listing 3.6: Parameters with sample values required to communicate with the OpenID provider

```
client_id: D4ZoMSpsxqgvUuiC6j5ROnEYea0a
redirect_uri: https://localhost:3000
Authorization endpoint: https://localhost:9443/oauth2/authz
Token endpoint: https://localhost:9443/oauth2/token
Issuer: https://localhost:9443
```

3.11.3 Updating the client application to use OpenID Connect login

In this section we are going to update the React application we developed in section 3.11.1 to support login with OpenID Connect. If you are already running the node server, which hosts the React application from section 3.11.1, please take it down by pressing Ctrl + C on the terminal that runs the node server.

To enable OpenID Connect login to the React application, in chapter03/sample01 directory we use the npm package `@facilelogin/oidc-react`. It's an open source npm package released under the MIT licenses. We developed this package by forking two open source node modules: https://github.com/auth0/auth0-react and https://github.com/auth0/auth0-spa-js. If you are interested in finding more details you can find the two forked repositories with our changes here: https://github.com/openidconnect-in-action/oidc-react and https://github.com/openidconnect-in-action/oidc-spa-js.

To install the @facilelogin/oidc-react package, run the following command from sample01 directory. Once the command runs successfully, you'll find a new entry added into the sample01/package.json file under the dependencies section with respect to the @facilelogin/oidc-react package.

```
\> npm install @facilelogin/oidc-react
```

The `@facilelogin/oidc-react` package introduces a new React component called `<OIDCProvider />` that carries the configuration corresponding to your OpenID provider. To add this component to your React application, on sample01/src/index.js, replace the existing call to the `ReactDOM.render()` function with the following. You also need to have an import statement to import `OIDCProvider` component from the `@facilelogin/oidc-react` package.

Listing 3.7: Rendering the <ODICProvider /> React component

```
import { OIDCProvider } from '@facilelogin/oidc-react';

ReactDOM.render(
  <OIDCProvider
    domain="localhost:9443"
    tokenEp="https://localhost:9443/oauth2/token"
    authzEp="https://localhost:9443/oauth2/authorize"
    clientId="D4ZoMSpsxqgvUuiC6j5ROnEYea0a"
    issuer="https://localhost:9443/oauth2/token"
    redirectUri={window.location.origin}>
    <App />
  </OIDCProvider>,
  document.getElementById('book-club-app')
);
```

Now you can replace the code in sample01/src/components/App.js with the following, that adds a login button to the welcome page.

Listing 3.8: Updated App.js code that renders the login button to initiate the login flow

```
import React from 'react';
import { useAuth0 } from '@facilelogin/oidc-react';

function App() {
  const {isLoading,isAuthenticated,error,user,loginWithRedirect,logout,} = useAuth0();

  if (isLoading) {
    return <div>Loading...</div>;
  }
  if (error) {
    return <div>Oops... {error.message}</div>;
  }

  if (isAuthenticated) {
    console.log(user.id);
    return (
      <div>
        Hello {user.sub}{' '}
        <button onClick={() => logout({ returnTo: window.location.origin })}>
          Log out
        </button>
      </div>
    );
  } else {
    return <button onClick={loginWithRedirect}>Log in</button>;
  }
}

export default App;
```

Now you can build the updated React application and start the node server using the following two npm commands.

```
\> npm run build
\> npm start
```

Once the node server successfully started up, you can visit http://localhost:3000 and click on the login button to initiate the login flow; and you will get redirected to the OpenID provider's login page.

3.12 Summary

- OpenID Connect defines three authentication flows: ***authorization code*** flow, ***implicit*** flow, and ***hybrid*** flow.
- An authentication flow in OpenID Connect uses OAuth 2.0 grant types, but an authentication flow is more than a grant type. It defines additional request/response parameters on top of what is already defined by OAuth 2.0 for grant types.
- The OpenID Connect specification defines the authentication flows in a self-contained manner in itself. So we should not confuse the OAuth 2.0 grant types with OpenID Connect authentication flows.
- The implicit authentication flow uses `id_token` or `id_token token` as the value of the `response_type` parameter in the authentication request and gets both the access token and the ID token from the authorization endpoint of the OpenID provider.
- The authorization code authentication flow uses code as the value of the `response_type` parameter in the authentication request and gets the access token and the ID token from the token endpoint of the OpenID provider.
- In practice, some SPAs use implicit flow as well as authorization code flow. However, most new SPAs use only authorization code flow.