Manually Build a Login Flow

Updated: Jun 30, 2026

Copy for LLM View as Markdown

For browser-based login for a web or desktop app without using our SDKs, such as in a webview for a native desktop app (for example Windows 8), or a login flow using entirely server-side code, you can build a Login flow for yourself by using browser redirects. This guide will take you through each step of the login flow and show you how to implement each one without using our SDKs:

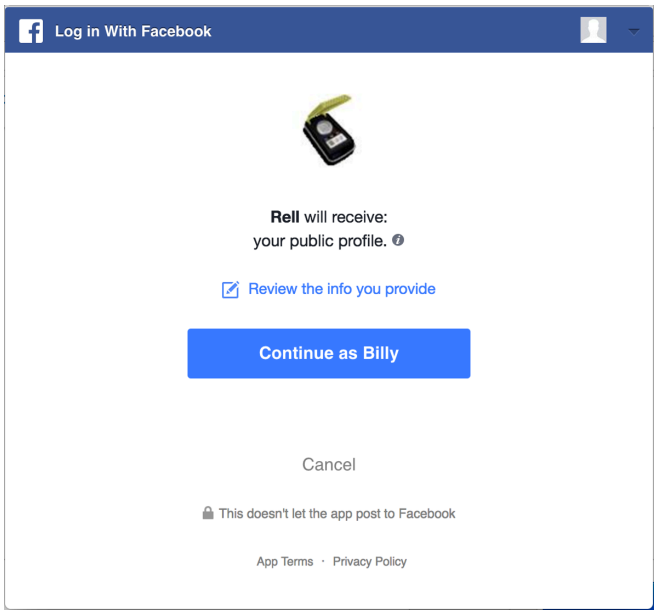
- Checking login status
- Logging people in
- Confirming identity
- Storing access tokens and login status
- Logging people out
- Detecting When People Uninstall Apps
- Responding to Requests to Delete User Data

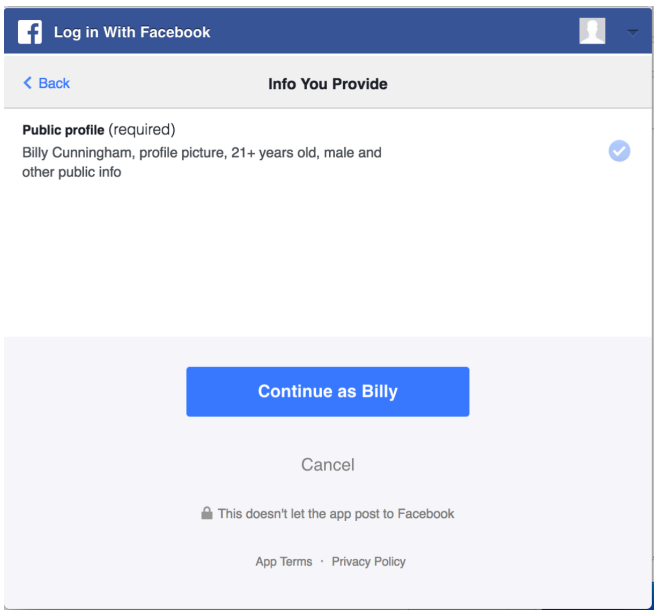
> To use Facebook Login in a desktop app, you'll need to be able to embed a web browser (sometimes called a webview) within the app to perform the login process.

Checking Login Status

Apps using our SDKs can check whether someone has already logged in using built-in functions. All other apps must create their own way of storing when a person has logged in, and when that indicator is not there, proceed on the assumption that they are logged out. If someone is logged out, then your app should redirect them to the Login dialog at an appropriate time — for example if they click a login button.

Whether someone is not logged into your app or not logged into Facebook, you can use the Login dialog to prompt them to do both. If they aren't logged into Facebook, they'll be prompted to login and then move onto logging into your app. This is automatically detected, so you don't need to do anything extra to enable this behavior.





Invoking the Login Dialog and Setting the Redirect URL

Your app must initiate a redirect to an endpoint which will display the login dialog:

```
https://www.facebook.com/v26.0/dialog/oauth?client_id={app-id}
&redirect_uri={redirect-uri}
&state={state-param}
```

This endpoint has the following required parameters:

- `client_id`. The ID of your app, found in your app's dashboard.
- `redirect_uri`. The URL that you want to redirect the person logging in back to. This URL will capture the response from the Login Dialog. If you are using this in a webview within a desktop app, this must be set to `https://www.facebook.com/connect/login_success.html`. You can confirm that this URL is set for your app in the App Dashboard. Under **Products** in the App Dashboard's left side navigation menu, click **Facebook Login**, then click **Settings**. Verify the **Valid OAuth redirect URIs** in the **Client OAuth Settings** section.

For example, if your login request looks like:

```
https://www.facebook.com/v26.0/dialog/oauth?client_id={app-id}
&redirect_uri={"https://www.domain.com/login"}
&state={"{st=state123abc,ds=123456789}"}
```

then your redirect URI would be called with this:

```
https://www.domain.com/login?state="{st=state123abc,ds=123456789}"
```

It also has the following optional parameters:

- `response_type`. Determines whether the response data included when the redirect back to the app occurs is in URL parameters or fragments. See the Confirming Identity section to choose which type your app should use. This can be one of:
- `code`. Response data is included as URL parameters and contains `code` parameter (an encrypted string unique to each login request). This is the default behavior if this parameter is not specified. It's most useful when your server will be handling the token.
- `token`. Response data is included as a URL fragment and contains an access token. Desktop apps must use this setting for `response_type`. This is most useful when the client will be handling the token.
- `code%20token`. Response data is included as a URL fragment and contains both an access token and the `code` parameter.
- `granted_scopes`. Returns a comma-separated list of all Permissions granted to the app by the user at the time of login. Can be combined with other `response_type` values. When combined with `token`, response data is included as a URL fragment, otherwise included as a URL parameter.
- `scope`. A comma or space separated list of Permissions to request from the person using your app.

For Windows 8 Apps

If you are building Login for a Windows app you can use the Package Security Identifier as your `redirect_uri`. Trigger the Login Dialog by calling `WebAuthenticationBroker.AuthenticateAsync` and use the Login Dialog endpoint as the requestUri. Here is an example in JavaScript:

```
    &response_type=token
    &redirect_uri=ms-app://{package-security-identifier}");

Windows.Security.Authentication.Web.WebAuthenticationBroker.authenticateAsync(
  options,
  requestUri)
  .done(function (result) {
    // Handle the response from the Login Dialog
  }
);
```

This will return control flow back to your app with an access token on success, or error on failure.

We recommend using the default browser for Windows 8 apps.

Handling Login Dialog Response

At this point in the login flow, the person will see the Login dialog and will have a choice of whether to cancel or to let the app access their data.

If the person using the app chooses OK on the Login dialog, they grant access to their public profile, friend list and any additional Permissions your app requested.

In all cases, the browser returns to the app, and response data indicating whether someone connected or cancelled is included. When your app uses the redirect method as above, the `redirect_uri` your app returns to will be appended with URL parameters or fragments (as per the chosen `response_type`), which must be captured.

Because of the various combinations of code languages that could be used in web apps, our guide doesn't show specific examples. However most modern languages will be capable of URL parsing, as follows:

Client-side JavaScript can capture URL fragments (for example jQuery BBQ), whereas URL parameters can be captured by both client-side and server-side code (for example `$_GET` in PHP, `jQuery.deparam` in jQuery BBQ, `querystring.parse` in Node.js or `urlparse` in Python). Microsoft provides a guide and sample code for Windows 8 apps connecting to an "online provider" - in this case, Facebook.

> When using a desktop app and logging in, Facebook redirects people to the `redirect_uri` mentioned above and places an access token along with some other metadata (such as token expiry time) in the URI fragment:

> Your app needs to detect this redirect and then read the access token out of the URI using the mechanisms provided by the OS and development framework you are using. You can then skip straight to the Inspecting access tokens step.

Canceled Login

If people using your app don't accept the Login dialog and clicks Cancel, they'll be redirected to the following:

```
YOUR_REDIRECT_URI?
 error_reason=user_denied
 &error=access_denied
 &error_description=Permissions+error.
```

See Handling Missing Permissions for more about what apps should do when people refuse to login.

Confirming Identity

Because this redirect flow involves browsers being redirected to URLs in your app from the Login dialog, traffic could directly access this URL with made-up fragments or parameters. If your app assumed these were valid parameters, the made-up data would be used by your app for potentially malicious purposes. As a result, your app should confirm that the person using the app is the same person that you have response data for before generating an access token for them. Confirming identity is accomplished in different ways depending on the `response_type` received above:

- When `code` is received, it has to be exchanged for an access token using an endpoint. The call will need to be server-to-server, since it involves your app secret. (Your app secret should never end up in client code.)
- When `token` is received, it needs to be verified. You should make an API call to an inspection endpoint that will indicate who the token was generated for and by which app. As this API call requires using an app access token, never make this call from a client. Instead make this call from a server where you can securely store your app secret.

to provide CSRF protection.

Exchanging Code for an Access Token

To get an access token, make an HTTP GET request to the following OAuth endpoint:

```
GET https://graph.facebook.com/v26.0/oauth/access_token?client_id={app-id}
&redirect_uri={redirect-uri}
&client_secret={app-secret}
&code={code-parameter}
```

This endpoint has some required parameters:

- `client_id`. Your app's IDs
- `redirect_uri`. This argument is required and must be the same as the original `request_uri` that you used when starting the OAuth login process.
- `client_secret`. Your unique app secret, shown on the App Dashboard. Never include this app secret in client-side code or in binaries that could be decompiled. It is extremely important that it remains completely secret as it is the core of the security of your app and all the people using it.
- `code`. The parameter received from the Login Dialog redirect above.

Response

The response you will receive from this endpoint will be returned in JSON format and, if successful, is:

```
{
  "access_token": {access-token},
  "token_type": {type},
  "expires_in":  {seconds-til-expiration}
}
```

If it is not successful, you will receive an explanatory error message.

Inspecting Access Tokens

Whether or not your app uses `code` or `token` as your `response_type` from the Login dialog, it will have received an access token. You can perform automated checks on these tokens using a Graph API endpoint:

This endpoint takes the following parameters:

- `input_token`. The token you need to inspect.
- `access_token` An app access token or an access token for a developer of the app.

The response of the API call is a JSON array containing data about the inspected token. For example:

```
{
    "data": {
        "app_id": 138483919580948,
        "type": "USER",
        "application": "Social Cafe",
        "expires_at": 1352419328,
        "is_valid": true,
        "issued_at": 1347235328,
        "metadata": {
            "sso": "iphone-safari"
        },
        "scopes": [
            "email",
            "publish_actions"
        ],
        "user_id": "1207059"
    }
}
```

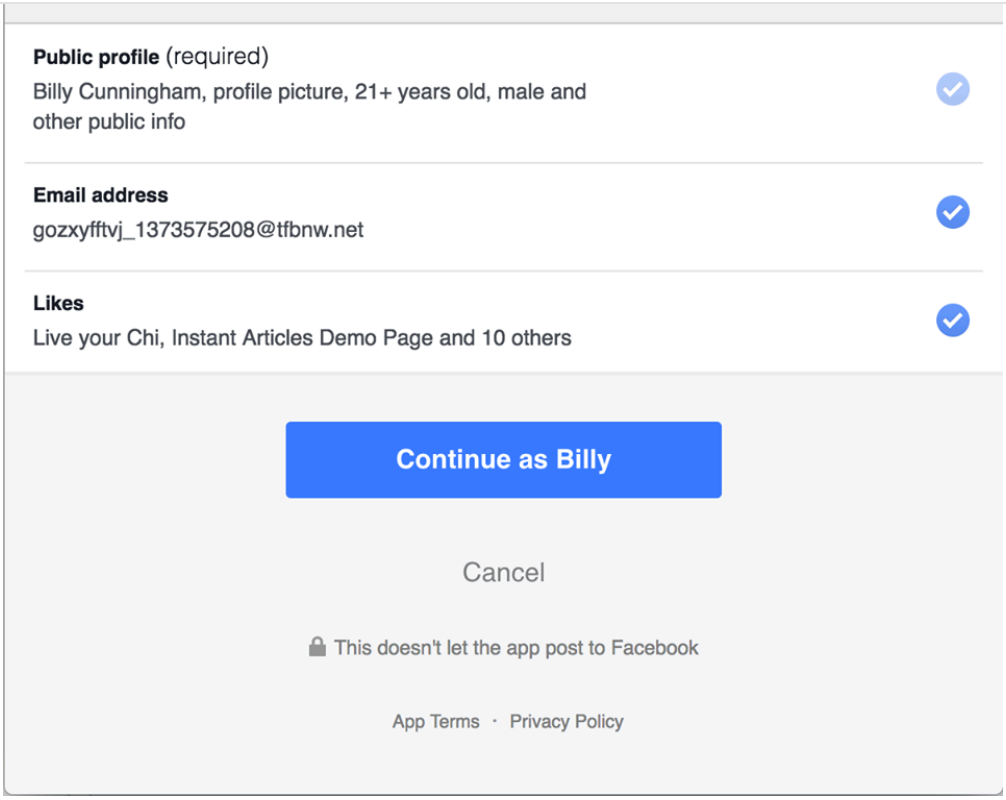
The `app_id` and `user_id` fields help your app verify that the access token is valid for the person and for your app. For a full description of the other fields, see the Getting Info about Access Tokens guide.

Checking Permissions

The `/me/permissions` edge can be called in order to retrieve a list of permissions that have been granted or declined by a particular user. Your app can use this to check which of the requested permissions it cannot use for any particular user.

Re-asking for Declined Permissions

Facebook Login lets people decline sharing some permissions with your app. The Login Dialog contains a screen that looks like this:



The `public_profile` permission is always required and greyed out because it can't be disabled.

However, if someone were to uncheck `user_likes` (Likes) in this example, checking `/me/permissions` for which permissions have been granted results in:

```
{
  "data":
    [
      {
        "permission":"public_profile",
        "status":"granted"
      },
      {
        "permission":"user_likes",
        "status":"declined"
      }
    ]
}
```

Note that `user_likes` has been declined instead of granted.

It's OK to ask a person again to grant your app permissions that they've declined. You should have a screen of education on why you think they should grant the permission to you and then re-ask. But if you invoke the Login Dialog as before, it won't ask for that permission.

You do this by adding the `auth_type=rerequest` parameter to your Login Dialog URL:

```
https://www.facebook.com/v26.0/dialog/oauth?client_id={app-id}
&redirect_uri={redirect-uri}
&auth_type=rerequest
scope=email
```

Using this, the Login Dialog will re-ask for the declined permission.

Storing Access Tokens and Login Status

At this point in the flow, you have someone authenticated and logged in. Your app is ready to make API calls on their behalf. Before doing so, it should store the access token and the login status of the person using the app.

Storing Access Tokens

After your app receives the access token from the previous step, the token should be stored so it's available to all parts of the app when it makes API calls. There is no specific process here, however in general if you're building a web app, it is best to add the token as a session variable to identify that browser session with a particular person, if you're building a native desktop or mobile app, then you should use the datastore available to your app. Also, the app should store the token in a database along with the `user_id` to identify it.

Please see our note about the size of access tokens in the access token document.

Tracking login status

Again, your app should store a person's login status, which helps avoid having to make additional calls to the Login dialog. Whatever procedure you chose, modify your login status checking to account for it.

Logging People Out

You can log people out of your app by undoing whatever login status indicator you added, for example deleting the session that indicates a person is logged in. You should also remove the stored access token.

Detecting When People Uninstall Apps

People are able to uninstall apps via Facebook.com without interacting with the app itself. To help apps detect when this has happened, we allow them to provide a de-authorize callback URL which will be pinged whenever this occurs.

You can enable a deauthorize callback through the App Dashboard.

Responding to Requests to Delete User Data

People can request an app to delete all information about them received from Facebook. For responding to these requests, see Data Deletion Request Callback.

English (US)