Authorizing OAuth apps

In this article

You can enable other users to authorize your OAuth app.

Note: Consider building a GitHub App instead of an OAuth app.

Both OAuth apps and GitHub Apps use OAuth 2.0.

GitHub Apps can act on behalf of a user, similar to an OAuth app, or as themselves, which is beneficial for automations that do not require user input. Additionally, GitHub Apps use fine grained permissions, give the user more control over which repositories the app can access, and use short-lived tokens. For more information, see "Differences between GitHub Apps and OAuth apps" and "About creating GitHub Apps."

GitHub Enterprise Server's OAuth implementation supports the standard authorization code grant type and the OAuth 2.0 Device Authorization Grant for apps that don't have access to a web browser.

If you want to skip authorizing your app in the standard way, such as when testing your app, you can use the non-web application flow.

To authorize your OAuth app, consider which authorization flow best fits your app.

- web application flow: Used to authorize users for standard OAuth apps that run in the browser. (The implicit grant type is not supported.)
- device flow: Used for headless apps, such as CLI tools.

Web application flow

Note: If you are building a GitHub App, you can still use the OAuth web application flow, but the setup has some important differences. See "Authenticating with a GitHub App on behalf of a user" for more information.

The web application flow to authorize users for your app is:

1. Users are redirected to request their GitHub identity
2. Users are redirected back to your site by GitHub
3. Your app accesses the API with the user's access token

1. Request a user's GitHub identity

```
GET http(s)://HOSTNAME/login/oauth/authorize
```

This endpoint takes the following input parameters.

Parameter name	Type	Description
`client_id`	`string`	**Required**. The client ID you received from GitHub when you registered.
`redirect_uri`	`string`	The URL in your application where users will be sent after authorization. See details below about redirect urls.
`login`	`string`	Suggests a specific account to use for signing in and authorizing the app.
`scope`	`string`	A space-delimited list of scopes. If not provided, `scope` defaults to an empty list for users that have not authorized any scopes for the application. For users who have authorized scopes for the application, the user won't be shown the OAuth authorization page with the list of scopes. Instead, this step of the flow will automatically complete with the set of scopes the user has authorized for the application. For example, if a user has already performed the web flow twice and has authorized one token with `user` scope and another token with `repo` scope, a third web flow that does not provide a `scope` will receive a token with `user` and `repo` scope.
`state`	`string`	An unguessable random string. It is used to protect against cross-site request forgery attacks.
`allow_signup`	`string`	Whether or not unauthenticated users will be offered an option to sign up for GitHub during the OAuth flow. The default is `true`. Use `false` when a policy prohibits signups.

The PKCE (Proof Key for Code Exchange) parameters `code_challenge` and `code_challenge_method` are not supported at this time. CORS pre-flight requests (OPTIONS) are not supported at this time.

2. Users are redirected back to your site by GitHub

If the user accepts your request, GitHub Enterprise Server redirects back to your site with a temporary `code` in a code parameter as well as the state you provided in the previous step in a `state` parameter. The temporary code will expire after 10 minutes. If the states don't match, then a third party created the request, and you should abort the process.

Exchange this `code` for an access token:

```
POST http(s)://HOSTNAME/login/oauth/access_token
```

This endpoint takes the following input parameters.

Parameter name	Type	Description
`client_id`	`string`	**Required.** The client ID you received from GitHub Enterprise Server for your OAuth app.
`client_secret`	`string`	**Required.** The client secret you received from GitHub Enterprise Server for your OAuth app.
`code`	`string`	**Required.** The code you received as a response to Step 1.
`redirect_uri`	`string`	The URL in your application where users are sent after authorization.

By default, the response takes the following form:

```
access_token=gho_16C7e42F292c6912E7710c838347Ae178B4a&scope=repo%2Cgist&token_type=
```

You can also receive the response in different formats if you provide the format in the `Accept` header. For example, `Accept: application/json` or `Accept: application/xml`:

```
Accept: application/json
{
  "access_token":"gho_16C7e42F292c6912E7710c838347Ae178B4a",
  "scope":"repo,gist",
  "token_type":"bearer"
}
```

```
Accept: application/xml
<OAuth>
  <token_type>bearer</token_type>
  <scope>repo,gist</scope>
  <access_token>gho_16C7e42F292c6912E7710c838347Ae178B4a</access_token>
</OAuth>
```

3. Use the access token to access the API

The access token allows you to make requests to the API on a behalf of a user.

```
Authorization: Bearer OAUTH-TOKEN
GET http(s)://HOSTNAME/api/v3/user
```

For example, in curl you can set the Authorization header like this:

```
curl -H "Authorization: Bearer OAUTH-TOKEN"
http(s)://<em>HOSTNAME</em>/api/v3/user
```

Device flow

> **Note:** The device flow is in public beta and subject to change.

The device flow allows you to authorize users for a headless app, such as a CLI tool or Git credential manager.

Before you can use the device flow to authorize and identify users, you must first enable it in your app's settings. For more information about enabling the device flow in your app, see "Modifying a GitHub App registration" for GitHub Apps and "Modifying an OAuth app" for OAuth apps.

Overview of the device flow

1. Your app requests device and user verification codes and gets the authorization URL where the user will enter the user verification code.

2. The app prompts the user to enter a user verification code at `http(s)://HOSTNAME/login/device`.

3. The app polls for the user authentication status. Once the user has authorized the device, the app will be able to make API calls with a new access token.

Step 1: App requests the device and user verification codes from GitHub

```
POST http(s)://HOSTNAME/login/device/code
```

Your app must request a user verification code and verification URL that the app will use to prompt the user to authenticate in the next step. This request also returns a device verification code that the app must use to receive an access token and check the status of user authentication.

The endpoint takes the following input parameters.

Parameter name	Type	Description
`client_id`	`string`	**Required.** The client ID you received from GitHub Enterprise Server for your app.
`scope`	`string`	A space-delimited list of the scopes that your app is requesting access to. For more information, see "Scopes for OAuth apps."

By default, the response takes the following form:

```
device_code=3584d83530557fdd1f46af8289938c8ef79f9dc5&expires_in=900&interval=5&user_
MJHT&verification_uri=https%3A%2F%HOSTNAME%2Flogin%2Fdevice
```

Parameter name	Type	Description
`device_code`	`string`	The device verification code is 40 characters and used to verify the device.
`user_code`	`string`	The user verification code is displayed on the device so the user can enter the code in a browser. This code is 8 characters with a hyphen in the middle.
`verification_uri`	`string`	The verification URL where users need to enter the `user_code`: `http(s)://HOSTNAME/login/device`.
`expires_in`	`integer`	The number of seconds before the `device_code` and `user_code` expire. The default is 900 seconds or 15 minutes.
`interval`	`integer`	The minimum number of seconds that must pass before you can make a new access token request (`POST http(s)://HOSTNAME/login/oauth/access_token`) to complete the device authorization. For example, if the interval is 5, then you cannot make a new request until 5 seconds pass. If you make more than one request over 5 seconds, then you will hit the rate limit and receive a `slow_down` error.

You can also receive the response in different formats if you provide the format in the `Accept` header. For example, `Accept: application/json` or `Accept: application/xml`:

```
Accept: application/json
{
  "device_code": "3584d83530557fdd1f46af8289938c8ef79f9dc5",
  "user_code": "WDJB-MJHT",
  "verification_uri": "http(s)://HOSTNAME/login/device",
  "expires_in": 900,
  "interval": 5
}
```

```
Accept: application/xml
<OAuth>
  <device_code>3584d83530557fdd1f46af8289938c8ef79f9dc5</device_code>
  <user_code>WDJB-MJHT</user_code>
  <verification_uri>http(s)://HOSTNAME/login/device</verification_uri>
  <expires_in>900</expires_in>
  <interval>5</interval>
```

```
</OAuth>
```

Step 2: Prompt the user to enter the user code in a browser

Your device will show the user verification code and prompt the user to enter the code at `http(s)://HOSTNAME/login/device`.

Step 3: App polls GitHub to check if the user authorized the device

```
POST http(s)://HOSTNAME/login/oauth/access_token
```

Your app will make device authorization requests that poll `POST http(s)://HOSTNAME/login/oauth/access_token`, until the device and user codes expire or the user has successfully authorized the app with a valid user code. The app must use the minimum polling `interval` retrieved in step 1 to avoid rate limit errors. For more information, see "Rate limits for the device flow."

The user must enter a valid code within 15 minutes (or 900 seconds). After 15 minutes, you will need to request a new device authorization code with `POST http(s)://HOSTNAME/login/device/code`.

Once the user has authorized, the app will receive an access token that can be used to make requests to the API on behalf of a user.

The endpoint takes the following input parameters.

Parameter name	Type	Description
`client_id`	`string`	**Required.** The client ID you received from GitHub Enterprise Server for your OAuth app.
`device_code`	`string`	**Required.** The device verification code you received from the `POST http(s)://HOSTNAME/login/device/code` request.
`grant_type`	`string`	**Required.** The grant type must be `urn:ietf:params:oauth:grant-type:device_code`.

By default, the response takes the following form:

```
access_token=gho_16C7e42F292c6912E7710c838347Ae178B4a&token_type=bearer&scope=repo%2
```

You can also receive the response in different formats if you provide the format in the `Accept` header. For example, `Accept: application/json` or `Accept: application/xml`:

```
Accept: application/json
{
 "access_token": "gho_16C7e42F292c6912E7710c838347Ae178B4a",
  "token_type": "bearer",
  "scope": "repo,gist"
}
```

```
Accept: application/xml
<OAuth>
  <access_token>gho_16C7e42F292c6912E7710c838347Ae178B4a</access_token>
  <token_type>bearer</token_type>
  <scope>gist,repo</scope>
</OAuth>
```

Rate limits for the device flow

When a user submits the verification code on the browser, there is a rate limit of 50 submissions in an hour per application.

If you make more than one access token request (`POST http(s)://HOSTNAME/login/oauth/access_token`) within the required minimum timeframe between requests (or `interval`), you'll hit the rate limit and receive a `slow_down` error response. The `slow_down` error response adds 5 seconds to the last `interval` . For more information, see the [Errors for the device flow](#).

Error codes for the device flow

Error code	Description
`authorization_pending`	This error occurs when the authorization request is pending and the user hasn't entered the user code yet. The app is expected to keep polling the `POST http(s)://HOSTNAME/login/oauth/access_token` request without exceeding the [`interval`](#) , which requires a minimum number of seconds between each request.
`slow_down`	When you receive the `slow_down` error, 5 extra seconds are added to the minimum `interval` or timeframe required between your requests using `POST http(s)://HOSTNAME/login/oauth/access_token` . For example, if the starting interval required at least 5 seconds between requests and you get a `slow_down` error response, you must now wait a minimum of 10 seconds before making a new request for an OAuth access token. The error response includes the new `interval` that you must use.
`expired_token`	If the device code expired, then you will see the `token_expired` error. You must make a new request for a device code.
`unsupported_grant_type`	The grant type must be `urn:ietf:params:oauth:grant-type:device_code` and included as an input parameter when you poll the OAuth token request `POST http(s)://HOSTNAME/login/oauth/access_token` .
`incorrect_client_credentials`	For the device flow, you must pass your app's client ID, which you can find on your app settings page. The `client_secret` is not needed for the device flow.
`incorrect_device_code`	The device_code provided is not valid.

`access_denied`	When a user clicks cancel during the authorization process, you'll receive a `access_denied` error and the user won't be able to use the verification code again.
`device_flow_disabled`	Device flow has not been enabled in the app's settings. For more information, see "Device flow."

For more information, see the "OAuth 2.0 Device Authorization Grant."

Non-Web application flow

Non-web authentication is available for limited situations like testing. If you need to, you can use Basic Authentication to create a personal access token using your personal access tokens settings page. This technique enables the user to revoke access at any time.

Redirect URLs

The `redirect_uri` parameter is optional. If left out, GitHub will redirect users to the callback URL configured in the OAuth app settings. If provided, the redirect URL's host (excluding sub-domains) and port must exactly match the callback URL. The redirect URL's path must reference a subdirectory of the callback URL.

```
CALLBACK: http://example.com/path

GOOD: http://example.com/path
GOOD: http://example.com/path/subdir/other
GOOD: http://oauth.example.com/path
GOOD: http://oauth.example.com/path/subdir/other
BAD:  http://example.com/bar
BAD:  http://example.com/
BAD:  http://example.com:8080/path
BAD:  http://oauth.example.com:8080/path
BAD:  http://example.org
```

Loopback redirect urls

The optional `redirect_uri` parameter can also be used for loopback URLs. If the application specifies a loopback URL and a port, then after authorizing the application users will be redirected to the provided URL and port. The `redirect_uri` does not need to match the port specified in the callback URL for the app.

For the `http://127.0.0.1:1234/path` callback URL, you can use this `redirect_uri`:

```
http://127.0.0.1:1234/path
```

Note that OAuth RFC recommends not to use `localhost`, but instead to use loopback literal `127.0.0.1` or IPv6 `::1`.

Creating multiple tokens for OAuth apps

You can create multiple tokens for a user/application/scope combination to create tokens for specific use cases.

This is useful if your OAuth app supports one workflow that uses GitHub for sign-in and only requires basic user information. Another workflow may require access to a user's private repositories. Using multiple tokens, your OAuth app can perform the web flow for each use case, requesting only the scopes needed. If a user only uses your application to sign in, they are never required to grant your OAuth app access to their private repositories.

There is a limit of ten tokens that are issued per user/application/scope combination, and a rate limit of ten tokens created per hour. If an application creates more than ten tokens for the same user and the same scopes, the oldest tokens with the same user/application/scope combination are revoked. However, hitting the hourly rate limit will not revoke your oldest token. Instead, it will trigger a re-authorization prompt within the browser, asking the user to double check the permissions they're granting your app. This prompt is intended to give a break to any potential infinite loop the app is stuck in, since there's little to no reason for an app to request ten tokens from the user within an hour.

Warning: Revoking all permission from an OAuth app deletes any SSH keys the application generated on behalf of the user, including deploy keys.

Directing users to review their access

You can link to authorization information for an OAuth app so that users can review and revoke their application authorizations.

To build this link, you'll need your OAuth app's `client_id` that you received from GitHub when you registered the application.

```
http(s)://HOSTNAME/settings/connections/applications/:client_id
```

Tip: To learn more about the resources that your OAuth app can access for a user, see "Discovering resources for a user."

Troubleshooting

- "Troubleshooting authorization request errors"
- "Troubleshooting OAuth app access token request errors"
- "Device flow errors"
- "Token expiration and revocation"

Further reading

- "About authentication to GitHub"