Best practices for creating a GitHub App

In this article

Follow these best practices to improve the security and performance of your GitHub App.

Select the minimum permissions required

When you register a GitHub App, select the minimum permissions that your GitHub App needs. If any keys or tokens for your app become compromised, this will limit the amount of damage that can occur. For more information about how to choose permissions, see "Choosing permissions for a GitHub App."

When your GitHub App creates an installation access token or user access token, you can further limit the repositories that the app can access and the permissions that the token has. For more information, see "Generating an installation access token for a GitHub App" and "Generating a user access token for a GitHub App."

Stay under the rate limit

Subscribe to webhook events instead of polling the API for data. This will help your GitHub App stay within the API rate limit. For more information, see "Using webhooks with GitHub Apps" and "Building a GitHub App that responds to webhook events."

Consider using conditional requests to help you stay within the rate limit. For more information about conditional requests, see "Resources in the REST API."

If possible, consider using consolidated GraphQL queries instead of REST API requests to help you stay within rate limits. For more information, see "About GitHub's APIs" and "GitHub GraphQL API documentation."

If you do hit a rate limit and need to retry an API request, use the `x-ratelimit-reset` or `Retry-After` response headers. If these headers are not available, wait for an exponentially increasing amount of time between retries, and throw an error after a specific number of retries. For more information, see "Best practices for using the REST API."

Secure your app's credentials

You can generate a private key and client secret for your GitHub App. With these credentials, your app can generate installation access tokens, user access tokens, and refresh tokens. These tokens can be used to make API requests on behalf of an app installation or user.

You must store these credentials securely. The storage mechanism depends on your integrations architecture and the platform that it runs on. In general, you should use a storage mechanism that is intended to store sensitive data on the platform that you are using.

Private keys

The private key for your GitHub App grants access to every account that the app is installed on.

Consider storing your GitHub App's private key in a key vault, such as Azure Key Vault, and making it sign-only.

Alternatively, you can store the key as an environment variable. However, this not as strong as storing the key in a key vault. If an attacker gains access to the environment, they can read the private key and gain persistent authentication as the GitHub App.

You should never hard code your private key in your app, even if your code is stored in a private repository. If your app is a native client, client-side app, or runs on a user device (as opposed to running on your servers), you should never ship your private key with your app.

You should not generate more private keys than you need. You should delete private keys that you no longer need. For more information, see "Managing private keys for GitHub Apps."

Client secrets

Client secrets are used to generate user access tokens for your app, unless your app uses device flow. For more information, see "Generating a user access token for a GitHub App."

If your app is a website or web app, consider storing your client secret in a key vault, such as Azure Key Vault, or as an encrypted environment variable or secret on your server.

If your app is a native client, client-side app, or runs on a user device (as opposed to running on your servers), you cannot secure your client secret. You should use caution if you plan to gate access to your own services based on tokens generated by your app because anyone can access the client secret to generate a token.

Installation access tokens, user access tokens, and refresh

tokens

Installation access tokens are used to make API requests on behalf of an app installation. User access tokens are used to make API requests on behalf of a user. Refresh tokens are used to regenerate user access tokens. Your app can use its private key to generate an installation access token. Your app can use its client secret to generate a user access token and refresh token.

If your app is a website or web app, you should encrypt the tokens on your back end and ensure there is security around the systems that can access the tokens. Consider storing refresh tokens in a separate place from active access tokens.

If your app is a native client, client-side app, or runs on a user device (as opposed to running on your servers), you may not be able to secure tokens as well as an app that runs on your servers. You should not generate installation access tokens since doing so requires a private key. Instead, you should generate user access tokens. You should store tokens via the mechanism recommended for your app's platform, and keep in mind that the storage mechanism may not be fully secure.

Use the appropriate token type

GitHub Apps can generate installation access tokens or user access tokens in order to make authenticated API requests.

Installation access tokens will attribute activity to your app. These are useful for automations that act independently of users.

User access tokens will attribute activity to a user and to your app. These are useful for taking actions based on user input or on behalf of a user.

An installation access token is restricted based on the GitHub App's permissions and access. A user access token is restricted based on both the GitHub App's permission and access and the user's permission and access. Therefore, if your GitHub App takes an action on behalf of a user, it should always use a user access token instead of an installation access token. Otherwise, your app might allow a user to see or do things that they shouldn't be able to see or do.

Your app should never use a personal access token or GitHub password to authenticate.

Validate organization access for every new authentication

When you use a user access token, you should track which organizations the token is authorized for. If an organization uses SAML SSO and a user has not performed SAML SSO, the user access token should not have access to that organization. You can use the `GET /user/installations` REST API endpoint to verify which organizations a user access token has access to. If the user is not authorized to access an organization, you should reject their access until they perform SAML SSO. For more information, see "GitHub App installations."

Expire tokens

GitHub strongly encourages you to use user access tokens that expire. If you previously opted out of using user access tokens that expire but want to re-enable this feature, see "Activating optional features for GitHub Apps."

Installation access tokens expire after one hour, expiring user access tokens expire after eight hours, and refresh tokens expire after six months. However, you can also revoke tokens as soon as you no longer need them. For more information, see "GitHub App installations" to revoke an installation access token and "OAuth Authorizations" to revoke a user access token.

Cache tokens

User access tokens and installation access tokens are meant to be used until they expire. You should cache tokens that you create. Before you create a new token, check your cache to see if you already have a valid token. Reusing tokens will make your app faster since it will make fewer requests to generate tokens.

Make a plan for handling security breaches

You should have a plan in place so that you can handle any security breaches in a timely manner.

In the event that your app's private key or secret is compromised, you will need to generate a new key or secret, update your app to use the new key or secret, and delete your old key or secret.

In the event that installation access tokens, user access tokens, or refresh tokens are compromised, you should immediately revoke these tokens. For more information, see "GitHub App installations" to revoke an installation access token and "OAuth Authorizations" to revoke a user access token.

Conduct regular vulnerability scans

You should conduct regular vulnerability scans for your app. For example, you might set up code scanning and secret scanning for the repository that hosts your app's code. For more information, see "About code scanning" and "About secret scanning."

Choose an appropriate environment

If your app runs on a server, verify that your server environment is secure and that it can handle the volume of traffic that you expect for your app.

Subscribe to the minimum webhooks

Only subscribe to the webhook events that your app needs. This will help reduce latency since your app won't be receiving payloads that it doesn't need.

Use a webhook secret

You should set a webhook secret for your GitHub App and verify that the signature of incoming webhook events match the secret. This helps to ensure that the incoming webhook event is a valid GitHub event.

For more information, see "Using webhooks with GitHub Apps." For an example, see "Building a GitHub App that responds to webhook events."

Allow time for users to accept new permissions

When you add repository or organization permissions to your GitHub App, users who have the app installed on their personal account or organization will receive an email prompting them to review the new permissions. Until the user approves the new permissions, their app installation will only receive the old permissions.

When you update permissions, you should consider making your app backwards compatible to give your users time to accept the new permissions. You can use the installation webhook with the `new_permissions_accepted` action property to learn when users accept new permissions for your app.

Use services in a secure manner

If your app uses third-party services, they should be used in a secure manner:

- Any services used by your app should have a unique login and password.
- Apps should not share service accounts such as email or database services to manage your SaaS service.
- Only employees with administrative duties should have admin access to the infrastructure that hosts your app.

Add logging and monitoring

Consider adding logging and monitoring capabilities for your app. A security log could include:

- Authentication and authorization events
- Service configuration changes
- Object reads and writes
- User and group permission changes
- Elevation of role to admin

Your logs should use consistent timestamping for each event and should record the users, IP addresses, or hostnames for all logged events.

Enable data deletion

If your GitHub App is available to other users or organizations, you should give users and organization owners a way to delete their data. Users should not need to email or call a support person in order to delete their data.

Further reading

- "Best practices for using webhooks"
- "Best practices for using the REST API"