Scripting with the REST API and Ruby

In this article

Learn how to write a script using the Octokit.rb SDK to interact with the REST API.

About Octokit.rb

If you want to write a script using Ruby to interact with the GitHub REST API, GitHub recommends that you use the Octokit.rb SDK. Octokit.rb is maintained by GitHub. The SDK implements best practices and makes it easier for you to interact with the REST API via Ruby. Octokit.rb works with all modern browsers, Node.rb, and Deno. For more information about Octokit.rb, see the Octokit.rb README.

Prerequisites

This guide assumes that you are familiar with Ruby and the GitHub REST API. For more information about the REST API, see "Getting started with the REST API."

You must install and import the `octokit` gem in order to use the Octokit.rb library. This guide uses import statements in accordance with Ruby's conventions. For more information about different installation methods, see the Octokit.rb README's Installation section.

Instantiating and authenticating

Warning: Treat your authentication credentials like a password.

To keep your credentials secure, you can store your credentials as a secret and run your script through GitHub Actions. For more information, see "Using secrets in GitHub Actions."

You can also store your credentials as a Codespaces secret and run your script in Codespaces. For more information, see "Managing secrets for your codespaces."

If these options are not possible, consider using another CLI service to store your credentials securely.

Authenticating with a personal access token

If you want to use the GitHub REST API for personal use, you can create a personal access token. For more information about creating a personal access token, see "Managing your personal access tokens."

First, require the `octokit` library. Then, create an instance of `Octokit` by passing your personal access token as the `access_token` option. In the following example, replace `YOUR-TOKEN` with your personal access token.

Ruby

```ruby
require 'octokit'

octokit = Octokit::Client.new(access_token: 'YOUR-TOKEN')
```

Authenticating with a GitHub App

If you want to use the API on behalf of an organization or another user, GitHub recommends that you use a GitHub App. If an endpoint is available to GitHub Apps, the REST reference documentation for that endpoint will say "Works with GitHub App." For more information, see "Registering a GitHub App," "About authentication with a GitHub App," and "Authenticating with a GitHub App on behalf of a user."

Instead of requiring `octokit`, create an instance of `Octokit::Client` by passing your GitHub App's information as options. In the following example, replace `APP_ID` with your app's ID, `PRIVATE_KEY` with your app's private key, and `INSTALLATION_ID` with the ID of the installation of your app that you want to authenticate on behalf of. You can find your app's ID and generate a private key on the settings page for your app. For more information, see "Managing private keys for GitHub Apps." You can get an installation ID with the `GET /users/{username}/installation`, `GET /repos/{owner}/{repo}/installation`, or `GET /orgs/{org}/installation` endpoints. For more information, see "GitHub Apps" in the REST reference documentation.

Ruby

```ruby
require 'octokit'

app = Octokit::Client.new(
  client_id: APP_ID,
  client_secret: PRIVATE_KEY,
  installation_id: INSTALLATION_ID
)

octokit = Octokit::Client.new(bearer_token:
app.create_app_installation.access_token)
```

Authenticating in GitHub Actions

If you want to use the API in a GitHub Actions workflow, GitHub recommends that you authenticate with the built-in `GITHUB_TOKEN` instead of creating a token. You can grant permissions to the `GITHUB_TOKEN` with the `permissions` key. For more information about `GITHUB_TOKEN`, see "Automatic token authentication."

If your workflow needs to access resources outside of the workflow's repository, then you will not be able to use `GITHUB_TOKEN`. In that case, store your credentials as a secret and replace `GITHUB_TOKEN` in the examples below with the name of your secret. For more information about secrets, see "Using secrets in GitHub Actions."

If you use the `run` keyword to execute your Ruby script in your GitHub Actions workflows, you can store the value of `GITHUB_TOKEN` as an environment variable. Your

script can access the environment variable as `ENV['VARIABLE_NAME']`.

For example, this workflow step stores `GITHUB_TOKEN` in an environment variable called `TOKEN`:

```
- name: Run script
  env:
    TOKEN: ${{ secrets.GITHUB_TOKEN }}
  run: |
    ruby .github/actions-scripts/use-the-api.rb
```

The script that the workflow runs uses `ENV['TOKEN']` to authenticate:

Ruby

```ruby
require 'octokit'

octokit = Octokit::Client.new(access_token: ENV['TOKEN'])
```

Instantiating without authentication

You can use the REST API without authentication, although you will have a lower rate limit and will not be able to use some endpoints. To create an instance of `Octokit` without authenticating, do not pass the `access_token` option.

Ruby

```ruby
require 'octokit'

octokit = Octokit::Client.new
```

Making requests

Octokit supports multiple ways of making requests. You can use the `request` method to make requests if you know the HTTP verb and path for the endpoint. You can use the `rest` method if you want to take advantage of autocompletion in your IDE and typing. For paginated endpoints, you can use the `paginate` method to request multiple pages of data.

Using the `request` method to make requests

To use the `request` method to make requests, pass the HTTP method and path as the first argument. Pass any body, query, or path parameters in a hash as the second argument. For example, to make a `GET` request to `/repos/{owner}/{repo}/issues` and pass the `owner`, `repo`, and `per_page` parameters:

Ruby

```ruby
octokit.request("GET /repos/{owner}/{repo}/issues", owner: "github", repo:
"docs", per_page: 2)
```

The `request` method automatically passes the `Accept: application/vnd.github+json` header. To pass additional headers or a different `Accept` header, add a `headers` option to the hash that is passed as a second argument. The value of the `headers` option is a hash with the header names as keys and header values as values. For example, to send

a `content-type` header with a value of `text/plain`:

```ruby
octokit.request("POST /markdown/raw", text: "Hello **world**", headers: {
"content-type" => "text/plain" })
```

Using `rest` endpoint methods to make requests

Every REST API endpoint has an associated `rest` endpoint method in Octokit. These methods generally autocomplete in your IDE for convenience. You can pass any parameters as a hash to the method.

```ruby
octokit.rest.issues.list_for_repo(owner: "github", repo: "docs", per_page: 2)
```

Making paginated requests

If the endpoint is paginated and you want to fetch more than one page of results, you can use the `paginate` method. `paginate` will fetch the next page of results until it reaches the last page and then return all of the results as an array. A few endpoints return paginated results as an array in an object, as opposed to returning the paginated results as an array. `paginate` always returns an array of items even if the raw result was an object.

For example, the following example gets all of the issues from the `github/docs` repository. Although it requests 100 issues at a time, the function won't return until the last page of data is reached.

```ruby
issue_data = octokit.paginate("GET /repos/{owner}/{repo}/issues", owner:
"github", repo: "docs", per_page: 100)
```

The `paginate` method accepts an optional block, which you can use to process each page of results. This allows you to collect only the data that you want from the response. For example, the following example continues to fetch results until an issue that includes "test" in the title is returned. For the pages of data that were returned, only the issue title and author are stored.

```ruby
issue_data = octokit.paginate("GET /repos/{owner}/{repo}/issues", owner:
"github", repo: "docs", per_page: 100) do |response, done|
  response.data.map do |issue|
    if issue.title.include?("test")
      done.call
    end
    { title: issue.title, author: issue.user.login }
  end
end
```

Instead of fetching all of the results at once, you can use `octokit.paginate.iterator()` to iterate through a single page at a time. For example, the following example fetches one page of results at a time and processes each object from the page before fetching

the next page. Once an issue that includes "test" in the title is reached, the script stops the iteration and returns the issue title and issue author of each object that was processed. The iterator is the most memory-efficient method for fetching paginated data.

Ruby

```ruby
iterator = octokit.paginate.iterator("GET /repos/{owner}/{repo}/issues", owner:
"github", repo: "docs", per_page: 100)
issue_data = []
break_loop = false
iterator.each do |data|
  break if break_loop
  data.each do |issue|
    if issue.title.include?("test")
      break_loop = true
      break
    else
      issue_data << { title: issue.title, author: issue.user.login }
    end
  end
end
```

You can use the `paginate` method with the `rest` endpoint methods as well. Pass the `rest` endpoint method as the first argument and any parameters as the second argument.

Ruby

```ruby
iterator = octokit.paginate.iterator(octokit.rest.issues.list_for_repo, owner:
"github", repo: "docs", per_page: 100)
```

For more information about pagination, see "Using pagination in the REST API."

Catching errors

Catching all errors

Sometimes, the GitHub REST API will return an error. For example, you will get an error if your access token is expired or if you omitted a required parameter. Octokit.rb automatically retries the request when it gets an error other than `400 Bad Request`, `401 Unauthorized`, `403 Forbidden`, `404 Not Found`, and `422 Unprocessable Entity`. If an API error occurs even after retries, Octokit.rb throws an error that includes the HTTP status code of the response (`response.status`) and the response headers (`response.headers`). You should handle these errors in your code. For example, you can use a try/catch block to catch errors:

Ruby

```ruby
begin
files_changed = []

iterator = octokit.paginate.iterator("GET
/repos/{owner}/{repo}/pulls/{pull_number}/files", owner: "github", repo: "docs",
pull_number: 22809, per_page: 100)
iterator.each do | data |
    files_changed.concat(data.map {
      | file_data | file_data.filename
    })
  end
rescue Octokit::Error => error
```

```
if error.response
puts "Error! Status: #{error.response.status}. Message: #
{error.response.data.message}"
end
puts error
end
```

Handling intended error codes

Sometimes, GitHub uses a 4xx status code to indicate a non-error response. If the endpoint you are using does this, you can add additional handling for specific errors. For example, the `GET /user/starred/{owner}/{repo}` endpoint will return a `404` if the repository is not starred. The following example uses the `404` response to indicate that the repository was not starred; all other error codes are treated as errors.

Ruby

```
begin
octokit.request("GET /user/starred/{owner}/{repo}", owner: "github", repo:
"docs")
puts "The repository is starred by me"
rescue Octokit::NotFound => error
puts "The repository is not starred by me"
rescue Octokit::Error => error
puts "An error occurred while checking if the repository is starred: #
{error&.response&.data&.message}"
end
```

Handling rate limit errors

If you receive a rate limit error, you may want to retry your request after waiting. When you are rate limited, GitHub responds with a `403 Forbidden` error, and the `x-ratelimit-remaining` response header value will be `"0"`. The response headers will include a `x-ratelimit-reset` header, which tells you the time at which the current rate limit window resets, in UTC epoch seconds. You can retry your request after the time specified by `x-ratelimit-reset`.

Ruby

```
def request_retry(route, parameters)
 begin
 response = octokit.request(route, parameters)
 return response
 rescue Octokit::RateLimitExceeded => error
 reset_time_epoch_seconds = error.response.headers['x-ratelimit-reset'].to_i
 current_time_epoch_seconds = Time.now.to_i
 seconds_to_wait = reset_time_epoch_seconds - current_time_epoch_seconds
 puts "You have exceeded your rate limit. Retrying in #{seconds_to_wait}
seconds."
 sleep(seconds_to_wait)
 retry
 rescue Octokit::Error => error
 puts error
 end
 end

 response = request_retry("GET /repos/{owner}/{repo}/issues", owner: "github",
repo: "docs", per_page: 2)
```

Using the response

The `request` method returns a response object if the request was successful. The response object contains `data` (the response body returned by the endpoint), `status` (the HTTP response code), `url` (the URL of the request), and `headers` (a hash containing the response headers). Unless otherwise specified, the response body is in JSON format. Some endpoints do not return a response body; in those cases, the `data` property is omitted.

Ruby

```ruby
response = octokit.request("GET /repos/{owner}/{repo}/issues/{issue_number}",
owner: "github", repo: "docs", issue_number: 11901)
 puts "The status of the response is: #{response.status}"
 puts "The request URL was: #{response.url}"
 puts "The x-ratelimit-remaining response header is: #{response.headers['x-
ratelimit-remaining']}"
 puts "The issue title is: #{response.data['title']}"
```

Similarly, the `paginate` method returns a response object. If the `request` was successful, the `response` object contains data, status, url, and headers.

Ruby

```ruby
response = octokit.paginate("GET /repos/{owner}/{repo}/issues", owner: "github",
repo: "docs", per_page: 100)
puts "#{response.data.length} issues were returned"
puts "The title of the first issue is: #{response.data[0]['title']}"
```

Example script

Here is a full example script that uses Octokit.rb. The script imports `Octokit` and creates a new instance of `Octokit`. If you want to authenticate with a GitHub App instead of a personal access token, you would import and instantiate `App` instead of `Octokit`. For more information, see "Authenticating with a GitHub App" in this guide.

The `get_changed_files` function gets all of the files changed for a pull request. The `comment_if_data_files_changed` function calls the `get_changed_files` function. If any of the files that the pull request changed include `/data/` in the file path, then the function will comment on the pull request.

Ruby

```ruby
require "octokit"

 octokit = Octokit::Client.new(access_token: "YOUR-TOKEN")

 def get_changed_files(octokit, owner, repo, pull_number)
 files_changed = []

 begin
 iterator = octokit.paginate.iterator("GET
/repos/{owner}/{repo}/pulls/{pull_number}/files", owner: owner, repo: repo,
pull_number: pull_number, per_page: 100)
 iterator.each do | data |
     files_changed.concat(data.map {
       | file_data | file_data.filename
     })
   end
 rescue Octokit::Error => error
 if error.response
 puts "Error! Status: #{error.response.status}. Message: #
```

```
{error.response.data.message}"
 end
 puts error
 end

 files_changed
 end

 def comment_if_data_files_changed(octokit, owner, repo, pull_number)
 changed_files = get_changed_files(octokit, owner, repo, pull_number)

 if changed_files.any ? {
   | file_name | /\/data\//i.match ? (file_name)
 }
 begin
 comment = octokit.create_pull_request_review_comment(owner, repo, pull_number,
"It looks like you changed a data file. These files are auto-generated. \n\nYou
must revert any changes to data files before your pull request will be
reviewed.")
 comment.html_url
 rescue Octokit::Error => error
 if error.response
 puts "Error! Status: #{error.response.status}. Message: #
{error.response.data.message}"
 end
 puts error
 end
 end
 end

# Example usage
owner = "github"
repo = "docs"
pull_number = 22809
comment_url = comment_if_data_files_changed(octokit, owner, repo, pull_number)

puts "A comment was added to the pull request: #{comment_url}"
```

Note: This is just a basic example. In practice, you may want to use error handling and conditional checks to handle various scenarios.

Next steps

To learn more about working with the GitHub REST API and Octokit.rb, explore the following resources:

- To learn more about Octokit.rb see the Octokit.rb documentation.
- To find detailed information about GitHub's available REST API endpoints, including their request and response structures, see the GitHub REST API documentation.