Scripting with the REST API and JavaScript

In this article

Write a script using the Octokit.js SDK to interact with the REST API.

About Octokit.js

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

Prerequisites

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

You must install and import `octokit` in order to use the Octokit.js library. This guide uses import statements in accordance with ES6. For more information about different installation and import methods, see the Octokit.js README's Usage 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."

If this is 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, import `Octokit` from `octokit`. Then, pass your personal access token when you create an instance of `Octokit`. In the following example, replace `YOUR-TOKEN` with a reference to your personal access token. Replace `HOSTNAME` with the name of your GitHub Enterprise Server instance.

JavaScript

```
import { Octokit } from "octokit";

const octokit = new Octokit({
  baseUrl: "http(s)://HOSTNAME/api/v3",
  auth: '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 Apps." 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 importing `Octokit` from `octokit`, import `App`. In the following example, replace `APP_ID` with a reference to your app's ID. Replace `PRIVATE_KEY` with a reference to your app's private key. Replace `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. Replace `HOSTNAME` with the name of your GitHub Enterprise Server instance.

JavaScript

```
import { App } from "octokit";

const app = new App({
  appId: APP_ID,
  privateKey: PRIVATE_KEY,
  Octokit: Octokit.defaults({
    baseUrl: "http(s)://HOSTNAME/api/v3",
  }),
});

const octokit = await app.getInstallationOctokit(INSTALLATION_ID);
```

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 JavaScript 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 `process.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: |
    node .github/actions-scripts/use-the-api.mjs
```

The script that the workflow runs uses `process.env.TOKEN` to authenticate:

JavaScript

```
import { Octokit } from "octokit";

const octokit = new Octokit({
  baseUrl: "http(s)://HOSTNAME/api/v3",
  auth: process.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 `auth` argument. Set the base URL to `http(s)://HOSTNAME/api/v3`. Replace `[hostname]` with the name of your GitHub Enterprise Server instance.

JavaScript

```
import { Octokit } from "octokit";

const octokit = new Octokit({
  baseUrl: "http(s)://HOSTNAME/api/v3",
});
```

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 an object 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:

JavaScript

```
await 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` property to the object that is passed as a second argument. The value of the `headers` property is an object 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` and a `x-github-api-version` header with a value of `2022-11-28`:

JavaScript

```
await octokit.request("POST /markdown/raw", {
  text: "Hello **world**",
  headers: {
    "content-type": "text/plain",
    "x-github-api-version": "2022-11-28",
  },
});
```

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 an object to the method.

JavaScript

```
await octokit.rest.issues.listForRepo({
  owner: "github",
  repo: "docs",
  per_page: 2
});
```

Additionally, if you are using a typed language such as TypeScript, you can import types to use with these methods. For more information, see the TypeScript section in the plugin-rest-endpoint-methods.js README.

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 a single array. A few endpoints return paginated results as 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.

JavaScript

```
const issueData = await octokit.paginate("GET /repos/{owner}/{repo}/issues", {
  owner: "github",
  repo: "docs",
  per_page: 100,
  headers: {
    "x-github-api-version": "2022-11-28",
  },
});
```

The `paginate` method accepts an optional map function, which you can use to collect only the data that you want from the response. This reduces memory usage by your script. The map function can take a second argument, `done`, which you can call to end the pagination before the last page is reached. This lets you fetch a subset of pages. 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.

JavaScript

```
const issueData = await octokit.paginate("GET /repos/{owner}/{repo}/issues", {
  owner: "github",
  repo: "docs",
  per_page: 100,
  headers: {
    "x-github-api-version": "2022-11-28",
  },
},
    (response, done) => response.data.map((issue) => {
    if (issue.title.includes("test")) {
      done()
    }
    return ({title: issue.title, author: issue.user.login})
  })
);
```

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.

JavaScript

```
const iterator = octokit.paginate.iterator("GET /repos/{owner}/{repo}/issues", {
  owner: "github",
  repo: "docs",
  per_page: 100,
  headers: {
    "x-github-api-version": "2022-11-28",
  },
});

let issueData = []
let breakLoop = false
for await (const {data} of iterator) {
  if (breakLoop) break
  for (const issue of data) {
    if (issue.title.includes("test")) {
      breakLoop = true
      break
    } else {
      issueData = [...issueData, {title: issue.title, author: issue.user.login}];
```

```
    }
  }
}
```

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

JavaScript

```
const iterator = octokit.paginate.iterator(octokit.rest.issues.listForRepo, {
  owner: "github",
  repo: "docs",
  per_page: 100,
  headers: {
    "x-github-api-version": "2022-11-28",
  },
});
```

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.js 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.js 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:

JavaScript

```
let filesChanged = []

try {
  const iterator = octokit.paginate.iterator("GET
/repos/{owner}/{repo}/pulls/{pull_number}/files", {
    owner: "github",
    repo: "docs",
    pull_number: 22809,
    per_page: 100,
    headers: {
      "x-github-api-version": "2022-11-28",
    },
  });

  for await (const {data} of iterator) {
    filesChanged = [...filesChanged, ...data.map(fileData => fileData.filename)];
  }
} catch (error) {
  if (error.response) {
    console.error(`Error! Status: ${error.response.status}. Message:
${error.response.data.message}`)
  }
  console.error(error)
}
```

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 errors codes are treated as errors.

JavaScript

```
try {
  await octokit.request("GET /user/starred/{owner}/{repo}", {
    owner: "github",
    repo: "docs",
    headers: {
      "x-github-api-version": "2022-11-28",
    },
  });

  console.log(`The repository is starred by me`);

} catch (error) {
  if (error.status === 404) {
    console.log(`The repository is not starred by me`);
  } else {
    console.error(`An error occurred while checking if the repository is starred:
${error?.response?.data?.message}`);
  }
}
```

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`.

JavaScript

```
async function requestRetry(route, parameters) {
  try {
    const response = await octokit.request(route, parameters);
    return response
  } catch (error) {
    if (error.response && error.status === 403 && error.response.headers['x-
ratelimit-remaining'] === '0') {
      const resetTimeEpochSeconds = error.response.headers['x-ratelimit-reset'];
      const currentTimeEpochSeconds = Math.floor(Date.now() / 1000);
      const secondsToWait = resetTimeEpochSeconds - currentTimeEpochSeconds;
      console.log(`You have exceeded your rate limit. Retrying in
${secondsToWait} seconds.`);
      setTimeout(requestRetry, secondsToWait * 1000, route, parameters);
    } else {
      console.error(error);
    }
  }
}

const response = await requestRetry("GET /repos/{owner}/{repo}/issues", {
    owner: "github",
    repo: "docs",
```

```
    per_page: 2
  })
```

Using the response

The `request` method returns a promise that resolves to an object if the request was successful. The object properties are `data` (the response body returned by the endpoint), `status` (the HTTP response code), `url` (the URL of the request), and `headers` (an object 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.

JavaScript

```
const response = await octokit.request("GET
/repos/{owner}/{repo}/issues/{issue_number}", {
  owner: "github",
  repo: "docs",
  issue_number: 11901,
  headers: {
    "x-github-api-version": "2022-11-28",
  },
});

console.log(`The status of the response is: ${response.status}`)
console.log(`The request URL was: ${response.url}`)
console.log(`The x-ratelimit-remaining response header is: ${response.headers["x-
ratelimit-remaining"]}`)
console.log(`The issue title is: ${response.data.title}`)
```

Similarly, the `paginate` method returns a promise. If the request was successful, the promise resolves to an array of data returned by the endpoint. Unlike the `request` method, the `paginate` method does not return the status code, URL, or headers.

JavaScript

```
const data = await octokit.paginate("GET /repos/{owner}/{repo}/issues", {
  owner: "github",
  repo: "docs",
  per_page: 100,
  headers: {
    "x-github-api-version": "2022-11-28",
  },
});

console.log(`${data.length} issues were returned`)
console.log(`The title of the first issue is: ${data[0].title}`)
```

Example script

Here is a full example script that uses Octokit.js. The script imports `Octokit` and creates a new instance of `Octokit`. If you wanted 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."

The `getChangedFiles` function gets all of the files changed for a pull request. The `commentIfDataFilesChanged` function calls the `getChangedFiles` 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.

JavaScript

```
import { Octokit } from "octokit";

const octokit = new Octokit({
  baseUrl: "http(s)://HOSTNAME/api/v3",
  auth: 'YOUR-TOKEN',
});

async function getChangedFiles({owner, repo, pullNumber}) {
  let filesChanged = []

  try {
    const iterator = octokit.paginate.iterator("GET
/repos/{owner}/{repo}/pulls/{pull_number}/files", {
      owner: owner,
      repo: repo,
      pull_number: pullNumber,
      per_page: 100,
      headers: {
        "x-github-api-version": "2022-11-28",
      },
    });

    for await (const {data} of iterator) {
      filesChanged = [...filesChanged, ...data.map(fileData =>
fileData.filename)];
    }
  } catch (error) {
    if (error.response) {
      console.error(`Error! Status: ${error.response.status}. Message:
${error.response.data.message}`)
    }
    console.error(error)
  }

  return filesChanged
}

async function commentIfDataFilesChanged({owner, repo, pullNumber}) {
  const changedFiles = await getChangedFiles({owner, repo, pullNumber});

  const filePathRegex = new RegExp(/\/data\//, "i");
  if (!changedFiles.some(fileName => filePathRegex.test(fileName))) {
    return;
  }

  try {
    const {data: comment} = await octokit.request("POST
/repos/{owner}/{repo}/issues/{issue_number}/comments", {
      owner: owner,
      repo: repo,
      issue_number: pullNumber,
      body: `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.`,
      headers: {
        "x-github-api-version": "2022-11-28",
      },
    });

    return comment.html_url;
  } catch (error) {
    if (error.response) {
      console.error(`Error! Status: ${error.response.status}. Message:
${error.response.data.message}`)
    }
    console.error(error)
  }
}
```

```
await commentIfDataFilesChanged({owner: "github", repo: "docs", pullNumber:
191});
```

Next steps

- To learn more about Octokit.js see the Octokit.js documentation.
- For some real life examples, look at how GitHub Docs uses Octokit.js by searching the GitHub Docs repository.