Getting started with the REST API

In this article

Learn how to use the GitHub REST API.

GitHub CLI curl JavaScript

About the GitHub REST API

This article describes how to use the GitHub REST API using GitHub CLI, JavaScript, or `curl`. For a quickstart guide, see "Quickstart for GitHub REST API."

When you make a request to the REST API, you will specify an HTTP method and a path. Additionally, you might also specify request headers and path, query, or body parameters. The API will return the response status code, response headers, and potentially a response body.

The REST API reference documentation describes the HTTP method, path, and parameters for every operation. It also displays example requests and responses for each operation. For more information, see the REST reference documentation.

For more information about GitHub's APIs, see "About GitHub's APIs."

Making a request

To make a request, first find the HTTP method and the path for the operation that you want to use. For example, the "Get Octocat" operation uses the `GET` method and the `/octocat` path. For the full reference documentation for this operation, see "Meta."

> **Note**: You must install GitHub CLI in order to use the commands in the GitHub CLI examples. For installation instructions, see the GitHub CLI repository.

If you are not already authenticated to GitHub CLI, you must use the `gh auth login` subcommand to authenticate before making any requests. For more information, see "Authenticating."

To make a request using GitHub CLI, use the `api` subcommand along with the path. Use the `--method` or `-X` flag to specify the method.

```
gh api /octocat --method GET
```

Note: You must install and import `octokit` in order to use the Octokit.js library used in the JavaScript examples. For more information, see the Octokit.js README.

To make a request using JavaScript, you can use Octokit.js. For more information, see "Scripting with the REST API and JavaScript."

First, create an instance of `Octokit`. Set the base URL to `http(s)://HOSTNAME/api/v3`. Replace `[hostname]` with the name of your GitHub Enterprise Server instance.

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

Then, use the `request` method to make requests. Pass the HTTP method and path as the first argument.

```
await octokit.request("GET /octocat", {});
```

Prepend the base URL for the GitHub REST API, `http(s)://HOSTNAME/api/v3`, to the path to get the full URL: `http(s)://HOSTNAME/api/v3/octocat`. Replace `[hostname]` with the name of your GitHub Enterprise Server instance.

Use the `curl` command in your command line. Use the `--request` or `-X` flag followed by the HTTP method. Use the `--url` flag followed by the full URL.

```
curl --request GET \
--url "https://api.github.com/octocat"
```

Note: If you get a message similar to "command not found: curl", you may need to download and install `curl`. For more information, see the curl project download page.

Continue reading to learn how to authenticate, send parameters, and use the response.

Authenticating

Many operations require authentication or return additional information if you are authenticated. Additionally, you can make more requests per hour when you are authenticated.

Although some REST API operations are accessible without authentication, you must authenticate to GitHub CLI in order to use the `api` subcommand.

About tokens

You can authenticate your request by adding a token.

If you want to use the GitHub REST API for personal use, you can create a personal access token. The REST API operations used in this article require `repo` scope for personal access tokens (classic) or, unless otherwise noted, read-only access to public repositories for fine-grained personal access tokens. Other operations may require different scopes or permissions. For more information about creating a personal access token, see "Managing your personal access tokens."

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 operation is available to GitHub Apps, the REST reference documentation for that operation will say "Works with GitHub Apps." The REST API operations used in this article require `issues` read and write permissions for GitHub Apps. Other operations may require different permissions. 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."

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, see "Automatic token authentication."

For more information about best practices you can use to keep your tokens secure, see "Keeping your API credentials secure."

Authentication example

With GitHub CLI, you don't need to create an access token in advance. Use the `auth login` subcommand to authenticate to GitHub CLI:

```
gh auth login
```

You can use the `--scopes` flag to specify what scopes you want. If you want to authenticate with a token that you created, you can use the `--with-token` flag. For more information, see the GitHub CLI `auth login` documentation.

Warning: Treat your access token like a password.

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

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

To authenticate with the Octokit.js library, you can pass your token when you create an instance of `Octokit`. Replace `YOUR-TOKEN` with your token. Replace `[hostname]` with the name of your GitHub Enterprise Server instance.

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

Warning: Treat your access token like a password.

To help keep your account secure, you can use GitHub CLI instead of `curl` commands. GitHub CLI will take care of authentication for you. For more information, see the GitHub CLI version of this page.

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

In `curl` commands, you will send an `Authorization` header with your token. Replace `YOUR-TOKEN` with your token:

```
curl --request GET \
--url "https://api.github.com/octocat" \
--header "Authorization: Bearer YOUR-TOKEN"
```

Note: In most cases, you can use `Authorization: Bearer` or `Authorization: token` to pass a

token. However, if you are passing a JSON web token (JWT), you must use `Authorization: Bearer`.

Authentication example for GitHub Actions

You can also use the `run` keyword to execute GitHub CLI commands in your GitHub Actions workflows. For more information, see "Workflow syntax for GitHub Actions."

Instead of using the `gh auth login` command, pass your token as an environment variable called `GH_TOKEN`. GitHub recommends that you authenticate with the built-in `GITHUB_TOKEN` instead of creating a token. If this is not possible, store your token as a secret and replace `GITHUB_TOKEN` in the example below with the name of your secret. For more information about `GITHUB_TOKEN`, see "Automatic token authentication." For more information about secrets, see "Using secrets in GitHub Actions."

```yaml
jobs:
  use_api:
    runs-on: ubuntu-latest
    permissions: {}
    steps:
      - env:
          GH_TOKEN: ${{ secrets.GITHUB_TOKEN }}
        run: |
          gh api /octocat
```

You can also use the `run` keyword to execute your JavaScript scripts in your GitHub Actions workflows. For more information, see "Workflow syntax for GitHub Actions."

GitHub recommends that you authenticate with the built-in `GITHUB_TOKEN` instead of creating a token. If this is not possible, store your token as a secret and replace `GITHUB_TOKEN` in the example below with the name of your secret. For more information about `GITHUB_TOKEN`, see "Automatic token authentication." For more information about secrets, see "Using secrets in GitHub Actions."

The following example workflow:

1. Checks out the repository content
2. Sets up Node.js
3. Installs `octokit`
4. Stores the value of `GITHUB_TOKEN` as an environment variable called `TOKEN` and runs `.github/actions-scripts/use-the-api.mjs`, which can access that environment variable as `process.env.TOKEN`

Example workflow:

```yaml
on:
  workflow_dispatch:
jobs:
  use_api_via_script:
    runs-on: ubuntu-latest
    permissions: {}
    steps:
      - name: Check out repo content
        uses: actions/checkout@v4

      - name: Setup Node
        uses: actions/setup-node@v3
        with:
          node-version: '16.17.0'
```

```
          cache: npm

      - name: Install dependencies
        run: npm install octokit

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

Example JavaScript script, with the file path `.github/actions-scripts/use-the-api.mjs`:

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

const octokit = new Octokit({
  baseUrl: "http(s)://HOSTNAME/api/v3",
  auth: process.env.TOKEN,
});

await octokit.request("GET /octocat", {});
```

Instead of storing your script in a separate file and executing the script from your workflow, you can use the `actions/github-script` action to run a script. For more information, see the actions/github-script README.

```
jobs:
  use_api_via_script:
    runs-on: ubuntu-latest
    permissions: {}
    steps:
      - uses: actions/github-script@v6
        with:
          github-token: ${{ secrets.GITHUB_TOKEN }}
          script: |
            await github.request('GET /octocat', {})
```

You can also use the `run` keyword to execute `curl` commands in your GitHub Actions workflows. For more information, see "Workflow syntax for GitHub Actions."

GitHub recommends that you authenticate with the built-in `GITHUB_TOKEN` instead of creating a token. If this is not possible, store your token as a secret and replace `GITHUB_TOKEN` in the example below with the name of your secret. For more information about `GITHUB_TOKEN`, see "Automatic token authentication." For more information about secrets, see "Using secrets in GitHub Actions."

```
jobs:
  use_api:
    runs-on: ubuntu-latest
    permissions: {}
    steps:
      - env:
          GH_TOKEN: ${{ secrets.GITHUB_TOKEN }}
        run: |
          curl --request GET \
          --url "https://api.github.com/octocat" \
          --header "Authorization: Bearer $GH_TOKEN"
```

Using headers

Most operations specify that you should pass an `Accept` header with a value of `application/vnd.github+json`. Other operations may specify that you should send a

different `Accept` header or additional headers.

To send a header with GitHub CLI, use the `--header` or `-H` flag followed by the header in `key: value` format.

```
gh api --header 'Accept: application/vnd.github+json' --header 'X-GitHub-Api-
Version:2022-11-28' --method GET /octocat
```

The Octokit.js library 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 to the `request` method. 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`:

```
await octokit.request("GET /octocat", {
  headers: {
    "content-type": "text/plain",
    "X-GitHub-Api-Version": "2022-11-28",
  },
});
```

To send a header in a `curl` command, use the `--header` or `-H` flag followed by the header in `key: value` format.

```
curl --request GET \
--url "https://api.github.com/octocat" \
--header "Accept: application/vnd.github+json" \
--header "Authorization: Bearer YOUR-TOKEN" \
--header "X-GitHub-Api-Version: 2022-11-28"
```

Using path parameters

Path parameters modify the operation path. For example, the "List repository issues" path is `/repos/{owner}/{repo}/issues`. The curly brackets `{}` denote path parameters that you need to specify. In this case, you must specify the repository owner and name. For the reference documentation for this operation, see "Issues."

Note: In order for this command to work for your GitHub Enterprise Server instance, replace `octocat/Spoon-Knife` with a repository owned by your GitHub Enterprise Server instance. Otherwise, rerun the `gh auth login` command to authenticate to GitHub.com instead of your GitHub Enterprise Server instance.

To get issues from the `octocat/Spoon-Knife` repository, replace `{owner}` with `octocat` and `{repo}` with `Spoon-Knife`.

```
gh api --header 'Accept: application/vnd.github+json' --method GET
/repos/octocat/Spoon-Knife/issues
```

Note: In order for this example to work for your GitHub Enterprise Server instance, replace `octocat/Spoon-Knife` with a repository owned by your GitHub Enterprise Server instance. Otherwise, create a new `Octokit` instance and do not specify `baseURL`.

When you make a request with Octokit.js, all parameters, including path parameters, are passed in an object as the second argument to the `request` method. To get issues from the `octocat/Spoon-Knife` repository, specify `owner` as `octocat` and `repo` as `Spoon-Knife`.

```
await octokit.request("GET /repos/{owner}/{repo}/issues", {
  owner: "octocat",
  repo: "Spoon-Knife"
});
```

To get issues from the `octocat/Spoon-Knife` repository, replace `{owner}` with `octocat` and `{repo}` with `Spoon-Knife`. To build the full path, prepend the base URL for the GitHub REST API, `https://api.github.com`: `https://api.github.com/repos/octocat/Spoon-Knife/issues`.

Note: If you want to use your GitHub Enterprise Server instance instead of GitHub.com, use `http(s)://HOSTNAME/api/v3` instead of `https://api.github.com` and replace `[hostname]` with the name of your GitHub Enterprise Server instance. Replace `octocat/Spoon-Knife` with a repository owned by your GitHub Enterprise Server instance.

```
curl --request GET \
--url "https://api.github.com/repos/octocat/Spoon-Knife/issues" \
--header "Accept: application/vnd.github+json" \
--header "Authorization: Bearer YOUR-TOKEN"
```

The operation returns a list of issues and data about each issue. For more information about using the response, see the "Using the response" section.

Using query parameters

Query parameters allow you to control what data is returned for a request. For example, a query parameter may let you specify how many items are returned when the response is paginated.

By default, the "List repository issues" operation returns thirty issues, sorted in descending order by the date they were created. You can use the `per_page` parameter to return two issues instead of 30. You can use the `sort` parameter to sort the issues by the date they were last updated instead of by the date they were created. You can use the `direction` parameter to sort the results in ascending order instead of descending order.

For GitHub CLI, use the `-F` flag to pass a parameter that is a number, Boolean, or null. Use `-f` to pass string parameters.

```
gh api --header 'Accept: application/vnd.github+json' --method GET
/repos/octocat/Spoon-Knife/issues -F per_page=2 -f sort=updated -f direction=asc
```

Some operations use query parameters that are arrays. To send an array in the query string, use the query parameter once per array item, and append `[]` after the query parameter name. For example, to provide an array of two repository IDs, use `-f repository_ids[]=REPOSITORY_A_ID -f repository_ids[]=REPOSITORY_B_ID`.

In order to use GitHub CLI with query parameters that are arrays, you must use GitHub CLI version 2.31.0 or greater. For upgrade instructions, see the GitHub CLI repository.

When you make a request with Octokit.js, all parameters, including query parameters, are passed in an object as the second argument to the `request` method.

```
await octokit.request("GET /repos/{owner}/{repo}/issues", {
  owner: "octocat",
  repo: "Spoon-Knife",
  per_page: 2,
```

```
  sort: "updated",
  direction: "asc",
});
```

For `curl` commands, add a `?` to the end of the path, then append your query parameter name and value in the form `parameter_name=value`. Separate multiple query parameters with `&`.

```
curl --request GET \
--url "https://api.github.com/repos/octocat/Spoon-Knife/issues?
per_page=2&sort=updated&direction=asc" \
--header "Accept: application/vnd.github+json" \
--header "Authorization: Bearer YOUR-TOKEN"
```

Some operations use query parameters that are arrays. To send an array in the query string, use the query parameter once per array item, and append `[]` after the query parameter name. For example, to provide an array of two repository IDs, use `?repository_ids[]=REPOSITORY_A_ID&repository_ids[]=REPOSITORY_B_ID`.

The operation returns a list of issues and data about each issue. For more information about using the response, see the "Using the response" section.

Using body parameters

Body parameters allow you to pass additional data to the API. For example, the "Create an issue" operation requires you to specify a title for the new issue. It also lets you specify other information, such as text to put in the issue body. For the full reference documentation for this operation, see "Issues."

The "Create an issue" operation uses the same path as the "List repository issues" operation in the examples above, but it uses a `POST` method instead of a `GET` method.

For GitHub CLI, use the `-F` flag to pass a parameter that is a number, Boolean, or null. Use `-f` to pass string parameters. If the parameter is an array, you must append `[]` to the parameter name.

> In order to use GitHub CLI with body parameters that are arrays, you must use GitHub CLI version 2.21.0 or greater. For upgrade instructions, see the GitHub CLI repository.

```
gh api --header 'Accept: application/vnd.github+json' --method POST
/repos/octocat/Spoon-Knife/issues -f title="Created with the REST API" -f
body="This is a test issue created by the REST API" -f "labels[]=bug"
```

> If you are using a fine-grained personal access token, you must replace `octocat/Spoon-Knife` with a repository that you own or that is owned by an organization that you are a member of. Your token must have access to that repository and have read and write permissions for repository issues. For more information about creating a repository, see "Create a repo." For more information about granting access and permissions to a fine-grained personal access token, see "Managing your personal access tokens."

When you make a request with Octokit.js, all parameters, including body parameters, are passed in an object as the second argument to the `request` method.

```
await octokit.request("POST /repos/{owner}/{repo}/issues", {
  owner: "octocat",
  repo: "Spoon-Knife",
  title: "Created with the REST API",
  body: "This is a test issue created by the REST API",
```

```
});
```

If you are using a fine-grained personal access token, you must replace `octocat/Spoon-Knife` with a repository that you own or that is owned by an organization that you are a member of. Your token must have access to that repository and have read and write permissions for repository issues. For more information about creating a repository, see "Create a repo." For more information about granting access and permissions to a fine-grained personal access token, see "Managing your personal access tokens."

For `curl` commands, use the `--data` flag to pass the body parameters in a JSON object.

```
curl --request POST \
--url "https://api.github.com/repos/octocat/Spoon-Knife/issues" \
--header "Accept: application/vnd.github+json" \
--header "Authorization: Bearer YOUR-TOKEN" \
--data '{
  "title": "Created with the REST API",
  "body": "This is a test issue created by the REST API"
}'
```

The operation creates an issue and returns data about the new issue. In the response, find the `html_url` of your issue and navigate to your issue in the browser. For more information about using the response, see the "Using the response" section.

Using the response

About the response code and headers

Every request will return an HTTP status code that indicates the success of the response. For more information about response codes, see the MDN HTTP response status code documentation.

Additionally, the response will include headers that give more details about the response. Headers that start with `X-` or `x-` are custom to GitHub. For example, the `x-ratelimit-remaining` and `x-ratelimit-reset` headers tell you how many requests you can make in a time period.

To view the status code and headers, use the `--include` or `--i` flag when you send your request.

For example, this request:

```
gh api --header 'Accept: application/vnd.github+json' --method GET
/repos/octocat/Spoon-Knife/issues -F per_page=2 --include
```

returns the response code and headers like:

```
HTTP/2.0 200 OK
Access-Control-Allow-Origin: *
Access-Control-Expose-Headers: ETag, Link, Location, Retry-After, X-GitHub-OTP,
X-RateLimit-Limit, X-RateLimit-Remaining, X-RateLimit-Used, X-RateLimit-Resource,
X-RateLimit-Reset, X-OAuth-Scopes, X-Accepted-OAuth-Scopes, X-Poll-Interval, X-
GitHub-Media-Type, X-GitHub-SSO, X-GitHub-Request-Id, Deprecation, Sunset
Cache-Control: private, max-age=60, s-maxage=60
Content-Security-Policy: default-src 'none'
Content-Type: application/json; charset=utf-8
Date: Thu, 04 Aug 2022 19:56:41 GMT
Etag: W/"a63dfbcfdb73621e9d2e89551edcf9856731ced534bd7f1e114a5da1f5f73418"
Link: <https://api.github.com/repositories/1300192/issues?per_page=1&page=2>;
rel="next", <https://api.github.com/repositories/1300192/issues?
```

```
per_page=1&page=14817>; rel="last"
Referrer-Policy: origin-when-cross-origin, strict-origin-when-cross-origin
Server: GitHub.com
Strict-Transport-Security: max-age=31536000; includeSubdomains; preload
Vary: Accept, Authorization, Cookie, X-GitHub-OTP, Accept-Encoding, Accept, X-
Requested-With
X-Accepted-Oauth-Scopes: repo
X-Content-Type-Options: nosniff
X-Frame-Options: deny
X-Github-Api-Version-Selected: 2022-08-09
X-Github-Media-Type: github.v3; format=json
X-Github-Request-Id: 1C73:26D4:E2E500:1EF78F4:62EC2479
X-Oauth-Client-Id: 178c6fc778ccc68e1d6a
X-Oauth-Scopes: gist, read:org, repo, workflow
X-Ratelimit-Limit: 15000
X-Ratelimit-Remaining: 14996
X-Ratelimit-Reset: 1659645499
X-Ratelimit-Resource: core
X-Ratelimit-Used: 4
X-Xss-Protection: 0
```

In this example, the response code is `200`, which indicates a successful request.

When you make a request with Octokit.js, the `request` method returns a promise. If the request was successful, the promise resolves to an object that includes the HTTP status code of the response (`status`) and the response headers (`headers`). If an error occurs, the promise resolves to an object that includes the HTTP status code of the response (`status`) and the response headers (`response.headers`).

You can use a `try/catch` block to catch an error if it occurs. For example, if the request in the following script is successful, the script will log the status code and the value of the `x-ratelimit-remaining` header. If the request was not successful, the script will log the status code, the value of the `x-ratelimit-remaining` header, and the error message.

```
try {
  const result = await octokit.request("GET /repos/{owner}/{repo}/issues", {
    owner: "octocat",
    repo: "Spoon-Knife",
    per_page: 2,
  });

  console.log(`Success! Status: ${result.status}. Rate limit remaining:
${result.headers["x-ratelimit-remaining"]}`)

} catch (error) {
  console.log(`Error! Status: ${error.status}. Rate limit remaining:
${error.headers["x-ratelimit-remaining"]}. Message:
${error.response.data.message}`)
}
```

To view the status code and headers, use the `--include` or `--i` flag when you send your request.

For example, this request:

```
curl --request GET \
--url "https://api.github.com/repos/octocat/Spoon-Knife/issues?per_page=2" \
--header "Accept: application/vnd.github+json" \
--header "Authorization: Bearer YOUR-TOKEN" \
--include
```

returns the response code and headers like:

```
HTTP/2 200
server: GitHub.com
```

```
date: Thu, 04 Aug 2022 20:07:51 GMT
content-type: application/json; charset=utf-8
cache-control: public, max-age=60, s-maxage=60
vary: Accept, Accept-Encoding, Accept, X-Requested-With
etag: W/"7fceb7e8c958d3ec4d02524b042578dcc7b282192e6c939070f4a70390962e18"
x-github-media-type: github.v3; format=json
link: <https://api.github.com/repositories/1300192/issues?
per_page=2&sort=updated&direction=asc&page=2>; rel="next",
<https://api.github.com/repositories/1300192/issues?
per_page=2&sort=updated&direction=asc&page=7409>; rel="last"
access-control-expose-headers: ETag, Link, Location, Retry-After, X-GitHub-OTP,
X-RateLimit-Limit, X-RateLimit-Remaining, X-RateLimit-Used, X-RateLimit-Resource,
X-RateLimit-Reset, X-OAuth-Scopes, X-Accepted-OAuth-Scopes, X-Poll-Interval, X-
GitHub-Media-Type, X-GitHub-SSO, X-GitHub-Request-Id, Deprecation, Sunset
access-control-allow-origin: *
strict-transport-security: max-age=31536000; includeSubdomains; preload
x-frame-options: deny
x-content-type-options: nosniff
x-xss-protection: 0
referrer-policy: origin-when-cross-origin, strict-origin-when-cross-origin
content-security-policy: default-src 'none'
x-ratelimit-limit: 15000
x-ratelimit-remaining: 14996
x-ratelimit-reset: 1659645535
x-ratelimit-resource: core
x-ratelimit-used: 4
accept-ranges: bytes
content-length: 4936
x-github-request-id: 14E0:4BC6:F1B8BA:208E317:62EC2715
```

In this example, the response code is `200`, which indicates a successful request.

About the response body

Many operations will return a response body. Unless otherwise specified, the response body is in JSON format. For example, this request returns a list of issues with data about each issue:

```
gh api --header 'Accept: application/vnd.github+json' --method GET
/repos/octocat/Spoon-Knife/issues -F per_page=2
```

```
await octokit.request("GET /repos/{owner}/{repo}/issues", {
  owner: "octocat",
  repo: "Spoon-Knife",
  per_page: 2,
});
```

```
curl --request GET \
--url "https://api.github.com/repos/octocat/Spoon-Knife/issues?per_page=2" \
--header "Accept: application/vnd.github+json" \
--header "Authorization: Bearer YOUR-TOKEN"
```

Unlike the GraphQL API where you specify what information you want, the REST API typically returns more information than you need. If desired, you can parse the response to pull out specific pieces of information.

For example, you can use `>` to redirect the response to a file:

```
gh api --header 'Accept: application/vnd.github+json' --method GET
/repos/octocat/Spoon-Knife/issues -F per_page=2 > data.json
```

Then you can use jq to get the title and author ID of each issue:

```
jq '.[] | {title: .title, authorID: .user.id}' data.json
```

The previous two commands return something like:

```
{
  "title": "Update index.html",
  "authorID": 10701255
}
{
  "title": "Edit index file",
  "authorID": 53709285
}
```

For more information about jq, see the jq documentation.

For example, you can get the title and author ID of each issue:

```
try {
  const result = await octokit.request("GET /repos/{owner}/{repo}/issues", {
    owner: "octocat",
    repo: "Spoon-Knife",
    per_page: 2,
  });

  const titleAndAuthor = result.data.map(issue => {title: issue.title, authorID:
issue.user.id})

  console.log(titleAndAuthor)

} catch (error) {
  console.log(`Error! Status: ${error.status}. Message:
${error.response.data.message}`)
}
```

For example, you can use `>` to redirect the response to a file:

```
curl --request GET \
--url "https://api.github.com/repos/octocat/Spoon-Knife/issues?per_page=2" \
--header "Accept: application/vnd.github+json" \
--header "Authorization: Bearer YOUR-TOKEN" > data.json
```

Then you can use jq to get the title and author ID of each issue:

```
jq '.[] | {title: .title, authorID: .user.id}' data.json
```

The previous two commands return something like:

```
{
  "title": "Update index.html",
  "authorID": 10701255
}
{
  "title": "Edit index file",
  "authorID": 53709285
}
```

For more information about jq, see the jq documentation.

Next steps

This article demonstrated how to list and create issues in a repository. For more practice, try to comment on an issue, edit the title of an issue, or close an issue. For more information about these operations, see "Issues" and "Issues."

For more information about the operations that you can use, see the REST reference documentation.