Resources in the REST API

In this article

Learn how to navigate the resources provided by the GitHub Enterprise Server API.

API version

Available resources may vary between REST API versions. You should use the `X-GitHub-Api-Version` header to specify an API version. For more information, see "API Versions."

Schema

The API is accessed from `http(s)://HOSTNAME/api/v3`. All data is sent and received as JSON.

```
$ curl -I http(s)://<em>HOSTNAME</em>/api/v3/users/octocat/orgs

> HTTP/2 200
> Server: nginx
> Date: Fri, 12 Oct 2012 23:33:14 GMT
> Content-Type: application/json; charset=utf-8
> ETag: "a00049ba79152d03380c34652f2cb612"
> X-GitHub-Media-Type: github.v3
> x-ratelimit-limit: 5000
> x-ratelimit-remaining: 4987
> x-ratelimit-reset: 1350085394
> X-GitHub-Enterprise-Version: 3.10.0
> Content-Length: 5
> Cache-Control: max-age=0, private, must-revalidate
> X-Content-Type-Options: nosniff
```

Blank fields are included as `null` instead of being omitted.

All timestamps return in UTC time, ISO 8601 format:

```
YYYY-MM-DDTHH:MM:SSZ
```

For more information about timezones in timestamps, see this section.

Summary representations

When you fetch a list of resources, the response includes a *subset* of the attributes for that resource. This is the "summary" representation of the resource. (Some attributes are computationally expensive for the API to provide. For performance reasons, the summary representation excludes those attributes. To obtain those attributes, fetch the "detailed" representation.)

Example: When you get a list of repositories, you get the summary representation of each repository. Here, we fetch the list of repositories owned by the octokit organization:

```
GET /orgs/octokit/repos
```

Detailed representations

When you fetch an individual resource, the response typically includes *all* attributes for that resource. This is the "detailed" representation of the resource. (Note that authorization sometimes influences the amount of detail included in the representation.)

Example: When you get an individual repository, you get the detailed representation of the repository. Here, we fetch the octokit/octokit.rb repository:

```
GET /repos/octokit/octokit.rb
```

The documentation provides an example response for each API method. The example response illustrates all attributes that are returned by that method.

Authentication

GitHub recommends that you create a token to authenticate to the REST API. For more information about which type of token to create, see "Authenticating to the REST API."

You can authenticate your request by sending a token in the `Authorization` header of your request:

```
curl --request GET \
--url "http(s)://HOSTNAME/api/v3/octocat" \
--header "Authorization: Bearer YOUR-TOKEN" \
--header "X-GitHub-Api-Version: 2022-11-28"
```

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`.

If you try to use a REST API endpoint without a token or with a token that has insufficient permissions, you will receive a `404 Not Found` or `403 Forbidden` response.

OAuth2 key/secret

Deprecation Notice: GitHub will discontinue authentication to the API using query parameters. Authenticating to the API should be done with HTTP basic authentication. For more information, including scheduled brownouts, see the blog post.

Authentication to the API using query parameters while available is no longer supported due to security concerns. Instead we recommend integrators move their access token, `client_id`, or `client_secret` in the header. GitHub will announce the removal of authentication by query parameters with advanced notice.

```
curl -u my_client_id:my_client_secret
'http(s)://<em>HOSTNAME</em>/api/v3/user/repos'
```

Using your `client_id` and `client_secret` does *not* authenticate as a user, it will only identify your OAuth app to increase your rate limit. Permissions are only granted to users, not applications, and you will only get back data that an unauthenticated user would see. Don't leak your OAuth app's client secret to your users.

You will be unable to authenticate using your OAuth2 key and secret while in private mode, and trying to authenticate will return `401 Unauthorized`. For more information, see "Enabling private mode".

Failed login limit

Authenticating with invalid credentials will return `401 Unauthorized`:

```
$ curl -I http(s)://<em>HOSTNAME</em>/api/v3 --header "Authorization: Bearer
INVALID-TOKEN"
> HTTP/2 401

> {
>   "message": "Bad credentials",
>   "documentation_url": "https://docs.github.com/enterprise/3.10/rest"
> }
```

After detecting several requests with invalid credentials within a short period, the API will temporarily reject all authentication attempts for that user (including ones with valid credentials) with `403 Forbidden`:

```
> HTTP/2 403
> {
>   "message": "Maximum number of login attempts exceeded. Please try again
later.",
>   "documentation_url": "https://docs.github.com/enterprise/3.10/rest"
> }
```

Parameters

Many API methods take optional parameters. For `GET` requests, any parameters not specified as a segment in the path can be passed as an HTTP query string parameter:

```
curl -i "http(s)://<em>HOSTNAME</em>/api/v3/repos/vmg/redcarpet/issues?
state=closed"
```

In this example, the 'vmg' and 'redcarpet' values are provided for the `:owner` and `:repo` parameters in the path while `:state` is passed in the query string.

For `POST`, `PATCH`, `PUT`, and `DELETE` requests, parameters not included in the URL should be encoded as JSON with a Content-Type of 'application/json':

```
curl -i --header "Authorization: Bearer YOUR-TOKEN" -d '{"scopes":
["repo_deployment"]}' http(s)://<em>HOSTNAME</em>/api/v3/authorizations
```

Root endpoint

You can issue a `GET` request to the root endpoint to get all the endpoint categories that the REST API supports:

```
$ curl -u USERNAME:PASSWORD http(s)://<em>HOSTNAME</em>/api/v3
```

GraphQL global node IDs

See the guide on "Using global node IDs" for detailed information about how to find `node_id`s via the REST API and use them in GraphQL operations.

Client errors

There are three possible types of client errors on API calls that receive request bodies:

1. Sending invalid JSON will result in a `400 Bad Request` response.

```
HTTP/2 400
Content-Length: 35

{"message":"Problems parsing JSON"}
```

2. Sending the wrong type of JSON values will result in a `400 Bad Request` response.

```
HTTP/2 400
Content-Length: 40

{"message":"Body should be a JSON object"}
```

3. Sending invalid fields will result in a `422 Unprocessable Entity` response.

```
HTTP/2 422
Content-Length: 149

{
  "message": "Validation Failed",
  "errors": [
    {
      "resource": "Issue",
      "field": "title",
      "code": "missing_field"
    }
  ]
}
```

All error objects have resource and field properties so that your client can tell what the problem is. There's also an error code to let you know what is wrong with the field.

Error code	Description
`missing`	A resource does not exist.
`missing_field`	A required field on a resource has not been set.
`invalid`	The formatting of a field is invalid. Review the documentation for more specific information.
`already_exists`	Another resource has the same value as this field. This can happen in resources that must have some unique key (such as label names).
`unprocessable`	The inputs provided were invalid.

Resources may also send custom validation errors (where `code` is `custom`). Custom errors will always have a `message` field describing the error, and most errors will also include a `documentation_url` field pointing to some content that might help you resolve the error.

HTTP redirects

The GitHub Enterprise Server REST API uses HTTP redirection where appropriate. Clients should assume that any request may result in a redirection. Receiving an HTTP redirection is *not* an error and clients should follow that redirect. Redirect responses will have a `Location` header field which contains the URI of the resource to which the client should repeat the requests.

A `301` status code indicates permanent redirection. The URI you used to make the request has been superseded by the one specified in the `Location` header field. This and all future requests to this resource should be directed to the new URI.

A `302` or `307` status code indicates temporary redirection. The request should be repeated verbatim to the URI specified in the `Location` header field but clients should continue to use the original URI for future requests.

Other redirection status codes may be used in accordance with the HTTP 1.1 spec.

HTTP verbs

Where possible, the GitHub Enterprise Server REST API strives to use appropriate HTTP verbs for each action. Note that HTTP verbs are case-sensitive.

Verb	Description
`HEAD`	Can be issued against any resource to get just the HTTP header info.
`GET`	Used for retrieving resources.
`POST`	Used for creating resources.
`PATCH`	Used for updating resources with partial JSON data. For instance, an Issue resource has `title` and `body` attributes. A `PATCH` request may accept one or more of the attributes to update the resource.
`PUT`	Used for replacing resources or collections. For `PUT` requests with no `body` attribute, be sure to

	For requests with no `body` attribute, be sure to set the `Content-Length` header to zero.
`DELETE`	Used for deleting resources.

Hypermedia

All resources may have one or more `*_url` properties linking to other resources. These are meant to provide explicit URLs so that proper API clients don't need to construct URLs on their own. It is highly recommended that API clients use these. Doing so will make future upgrades of the API easier for developers. All URLs are expected to be proper RFC 6570 URI templates.

You can then expand these templates using something like the uri_template gem:

```
>> tmpl = URITemplate.new('/notifications{?since,all,participating}')
>> tmpl.expand
=> "/notifications"

>> tmpl.expand all: 1
=> "/notifications?all=1"

>> tmpl.expand all: 1, participating: 1
=> "/notifications?all=1&participating=1"
```

Timeouts

If GitHub takes more than 10 seconds to process an API request, GitHub will terminate the request and you will receive a timeout response like this:

```
{
    "message": "Server Error"
}
```

GitHub Enterprise Server reserves the right to change the timeout window to protect the speed and reliability of the API.

Rate limiting

The GitHub Enterprise Server REST API uses rate limiting to control API traffic. Different types of API requests have different rate limits. The response headers describe your current rate limit status.

Rate limits

Different types of API requests to your GitHub Enterprise Server instance are subject to different rate limits. Additionally, the Search endpoints have dedicated limits. For more information, see "Search" in the REST API documentation.

> Rate limits are disabled by default for GitHub Enterprise Server. Contact your site administrator to confirm the rate limits for your instance.

Rate limits for requests from personal accounts

You can make direct API requests that you authenticate with a personal access token. An

OAuth app or GitHub App can also make a request on your behalf after you authorize the app. For more information, see "Managing your personal access tokens," "Authorizing OAuth apps," and "Authorizing GitHub Apps."

GitHub Enterprise Server associates all of these requests with the authenticated user. For OAuth apps and GitHub Apps, this is the user who authorized the app. All of these requests count toward the authenticated user's rate limit.

By default, user access token requests are limited to 5,000 requests per hour and per authenticated user. All requests from OAuth apps authorized by a user or a personal access token owned by the user, and requests authenticated with any of the user's authentication credentials, share the same quota of 5,000 requests per hour for that user.

Rate limits for requests from GitHub Apps

Requests from a GitHub App may either use a user access token or an installation access token. For more information about rate limits for GitHub Apps, see "Rate limits for GitHub Apps."

Rate limits for requests from GitHub Actions

You can use the built-in `GITHUB_TOKEN` to authenticate requests in GitHub Actions workflows. For more information, see "Automatic token authentication."

When using `GITHUB_TOKEN`, the rate limit is 1,000 requests per hour per repository.

Checking your rate limit status

The response headers describe your current rate limit status. You can also use the REST API to find the current number of API calls available to you or your app at any given time.

Rate limit headers

The `x-ratelimit` response headers describe your current rate limit status following every request:

```
$ curl -i http(s)://<em>HOSTNAME</em>/api/v3/users/octocat
> HTTP/2 200
> x-ratelimit-limit: 60
> x-ratelimit-remaining: 56
> x-ratelimit-used: 4
> x-ratelimit-reset: 1372700873
```

Header name	Description
`x-ratelimit-limit`	The maximum number of requests you're permitted to make per hour.
`x-ratelimit-remaining`	The number of requests remaining in the current rate limit window.
`x-ratelimit-used`	The number of requests you've made in the current rate limit window.
`x-ratelimit-reset`	The time at which the current rate limit window resets in UTC epoch seconds.

Checking your rate limit status with the REST API

You can use the REST API to check your rate limit status without incurring a hit to the current limit. For more information, see "Rate limit." When possible, GitHub recommends using the `x-ratelimit` response headers instead to decrease load on the API.

Exceeding the rate limit

If you exceed the rate limit, the response will have a `403` status and the `x-ratelimit-remaining` header will be `0`:

```
> HTTP/2 403
> Date: Tue, 20 Aug 2013 14:50:41 GMT
> x-ratelimit-limit: 60
> x-ratelimit-remaining: 0
> x-ratelimit-used: 60
> x-ratelimit-reset: 1377013266

> {
>    "message": "API rate limit exceeded for xxx.xxx.xxx.xxx. (But here's the
good news: Authenticated requests get a higher rate limit. Check out the
documentation for more details.)",
>    "documentation_url":
"https://docs.github.com/enterprise/3.10/rest/overview/resources-in-the-rest-
api#rate-limiting"
> }
```

If you are rate limited, you should not try your request until after the time specified by the `x-ratelimit-reset` time.

Increasing the unauthenticated rate limit for OAuth apps

If your OAuth app needs to make unauthenticated calls to public resources at a higher rate limit, you can pass your app's client ID and secret before the endpoint route.

```
$ curl -u my_client_id:my_client_secret -I
http(s)://<em>HOSTNAME</em>/api/v3/meta
> HTTP/2 200
> Date: Mon, 01 Jul 2013 17:27:06 GMT
> x-ratelimit-limit: 5000
> x-ratelimit-remaining: 4966
> x-ratelimit-used: 34
> x-ratelimit-reset: 1372700873
```

Note: Never share your client secret with anyone or include it in client-side browser code.

Staying within the rate limit

If you exceed your rate limit using Basic Authentication or OAuth, you can likely fix the issue by caching API responses and using conditional requests.

Secondary rate limits

The rate limits described above apply to the entire REST API and are per-user or per-app. In order to provide quality service on GitHub Enterprise Server, additional rate limits may apply to some actions when using the API. For example, using the API to rapidly create content, poll aggressively instead of using webhooks, make multiple concurrent requests, or repeatedly request data that is computationally expensive may result in additional rate limiting.

These additional rate limits are not intended to interfere with legitimate use of the API. Your normal rate limits should be the only limit you target. To ensure you're acting as a good API citizen, check out our Best Practices guidelines.

If your application triggers an additional rate limit, you'll receive an informative response:

```
> HTTP/2 403
> Content-Type: application/json; charset=utf-8
> Connection: close

> {
>   "message": "You have exceeded a secondary rate limit and have been
temporarily blocked from content creation. Please retry your request again
later.",
>   "documentation_url":
"https://docs.github.com/enterprise/3.10/rest/overview/resources-in-the-rest-
api#secondary-rate-limits"
> }
```

You should wait and try your request at a later time. If the `retry-after` response header is present, you should not retry your request until after that many seconds has elapsed. If the `x-ratelimit-remaining` header is `0`, you should not retry your request until after the time, in UTC epoch seconds, specified by the `x-ratelimit-reset` header. Otherwise, wait for an exponentially increasing amount of time between retries, and throw an error after a specific number of retries.

Conditional requests

Most responses return an `ETag` header. Many responses also return a `Last-Modified` header. You can use the values of these headers to make subsequent requests to those resources using the `If-None-Match` and `If-Modified-Since` headers, respectively. If the resource has not changed, the server will return a `304 Not Modified`.

```
$ curl -I http(s)://<em>HOSTNAME</em>/api/v3/user
> HTTP/2 200
> Cache-Control: private, max-age=60
> ETag: "644b5b0155e6404a9cc4bd9d8b1ae730"
> Last-Modified: Thu, 05 Jul 2012 15:31:30 GMT
> Vary: Accept, Authorization, Cookie
> x-ratelimit-limit: 5000
> x-ratelimit-remaining: 4996
> x-ratelimit-reset: 1372700873

$ curl -I http(s)://<em>HOSTNAME</em>/api/v3/user -H 'If-None-Match:
"644b5b0155e6404a9cc4bd9d8b1ae730"'
> HTTP/2 304
> Cache-Control: private, max-age=60
> ETag: "644b5b0155e6404a9cc4bd9d8b1ae730"
> Last-Modified: Thu, 05 Jul 2012 15:31:30 GMT
> Vary: Accept, Authorization, Cookie
> x-ratelimit-limit: 5000
> x-ratelimit-remaining: 4996
> x-ratelimit-reset: 1372700873

$ curl -I http(s)://<em>HOSTNAME</em>/api/v3/user -H "If-Modified-Since: Thu, 05
Jul 2012 15:31:30 GMT"
> HTTP/2 304
> Cache-Control: private, max-age=60
> Last-Modified: Thu, 05 Jul 2012 15:31:30 GMT
> Vary: Accept, Authorization, Cookie
> x-ratelimit-limit: 5000
> x-ratelimit-remaining: 4996
> x-ratelimit-reset: 1372700873
```

Cross origin resource sharing

The API supports Cross Origin Resource Sharing (CORS) for AJAX requests from any origin. You can read the CORS W3C Recommendation, or this intro from the HTML 5 Security Guide.

Here's a sample request sent from a browser hitting `http://example.com`:

```
$ curl -I http(s)://<em>HOSTNAME</em>/api/v3 -H "Origin: http://example.com"
HTTP/2 302
Access-Control-Allow-Origin: *
Access-Control-Expose-Headers: ETag, Link, X-GitHub-OTP, x-ratelimit-limit, x-
ratelimit-remaining, x-ratelimit-reset, X-OAuth-Scopes, X-Accepted-OAuth-Scopes,
X-Poll-Interval
```

This is what the CORS preflight request looks like:

```
$ curl -I http(s)://<em>HOSTNAME</em>/api/v3 -H "Origin: http://example.com" -X
OPTIONS
HTTP/2 204
Access-Control-Allow-Origin: *
Access-Control-Allow-Headers: Authorization, Content-Type, If-Match, If-Modified-
Since, If-None-Match, If-Unmodified-Since, X-GitHub-OTP, X-Requested-With
Access-Control-Allow-Methods: GET, POST, PATCH, PUT, DELETE
Access-Control-Expose-Headers: ETag, Link, X-GitHub-OTP, x-ratelimit-limit, x-
ratelimit-remaining, x-ratelimit-reset, X-OAuth-Scopes, X-Accepted-OAuth-Scopes,
X-Poll-Interval
Access-Control-Max-Age: 86400
```

JSON-P callbacks

You can send a `?callback` parameter to any GET call to have the results wrapped in a JSON function. This is typically used when browsers want to embed GitHub Enterprise Server content in web pages by getting around cross domain issues. The response includes the same data output as the regular API, plus the relevant HTTP Header information.

```
$ curl http(s)://<em>HOSTNAME</em>/api/v3?callback=foo

> /**/foo({
>   "meta": {
>     "status": 200,
>     "x-ratelimit-limit": "5000",
>     "x-ratelimit-remaining": "4966",
>     "x-ratelimit-reset": "1372700873",
>     "Link": [ // pagination headers and other links
>       ["http(s)://<em>HOSTNAME</em>/api/v3?page=2", {"rel": "next"}]
>     ]
>   },
>   "data": {
>     // the data
>   }
> })
```

You can write a JavaScript handler to process the callback. Here's a minimal example you can try out:

```
<html>
<head>
<script type="text/javascript">
function foo(response) {
  var meta = response.meta;
  var data = response.data;
  console.log(meta);
  console.log(data);
}

var script = document.createElement('script');
script.src = 'http(s)://HOSTNAME/api/v3?callback=foo';

document.getElementsByTagName('head')[0].appendChild(script);
</script>
</head>

<body>
  <p>Open up your browser's console.</p>
</body>
</html>
```

All of the headers are the same String value as the HTTP Headers with one notable exception: Link. Link headers are pre-parsed for you and come through as an array of `[url, options]` tuples.

A link that looks like this:

```
Link: <url1>; rel="next", <url2>; rel="foo"; bar="baz"
```

... will look like this in the Callback output:

```
{
  "Link": [
    [
      "url1",
      {
        "rel": "next"
      }
    ],
    [
      "url2",
      {
        "rel": "foo",
        "bar": "baz"
      }
    ]
  ]
}
```

Timezones

Some requests that create new data, such as creating a new commit, allow you to provide time zone information when specifying or generating timestamps. We apply the following rules, in order of priority, to determine timezone information for such API calls.

- Explicitly providing an ISO 8601 timestamp with timezone information
- Using the `Time-Zone` header
- Using the last known timezone for the user
- Defaulting to UTC without other timezone information

Note that these rules apply only to data passed to the API, not to data returned by the

API. As mentioned in "Schema," timestamps returned by the API are in UTC time, ISO 8601 format.

Explicitly providing an ISO 8601 timestamp with timezone information

For API calls that allow for a timestamp to be specified, we use that exact timestamp. An example of this is the API to manage commits. For more information, see "Git database."

These timestamps look something like `2014-02-27T15:05:06+01:00`. Also see this example for how these timestamps can be specified.

Using the `Time-Zone` header

It is possible to supply a `Time-Zone` header which defines a timezone according to the list of names from the Olson database.

```
curl -H "Time-Zone: Europe/Amsterdam" -X POST
http(s)://<em>HOSTNAME</em>/api/v3/repos/github-
linguist/linguist/contents/new_file.md
```

This means that we generate a timestamp for the moment your API call is made in the timezone this header defines. For example, the API to manage contents generates a git commit for each addition or change and uses the current time as the timestamp. For more information, see "Repositories." This header will determine the timezone used for generating that current timestamp.

Using the last known timezone for the user

If no `Time-Zone` header is specified and you make an authenticated call to the API, we use the last known timezone for the authenticated user. The last known timezone is updated whenever you browse the GitHub Enterprise Server website.

Defaulting to UTC without other timezone information

If the steps above don't result in any information, we use UTC as the timezone to create the git commit.

Legal

© 2023 GitHub, Inc. Terms Privacy Status Pricing Expert services Blog