Forming calls with GraphQL

In this article

Learn how to authenticate to the GraphQL API, then learn how to create and run queries and mutations.

Authenticating with GraphQL

You can authenticate to the GraphQL API using a personal access token, GitHub App, or OAuth app.

Authenticating with a personal access token

To authenticate with a personal access token, follow the steps in "Managing your personal access tokens." The data that you are requesting will dictate which scopes or permissions you will need.

For example, select the "issues:read" permission to read all of the issues in the repos your token has access to.

All fine-grained personal access tokens include read access to public repositories. To access public repositories with a personal access token (classic), select the "public_repo" scope.

If your token does not have the required scopes or permissions to access a resource, the API will return an error message that states the scopes or permissions your token needs.

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. In order to attribute activity to your app, you can make your app authenticate as an app installation. In order to attribute app activity to a user, you can make your app authenticate on behalf of a user. In both cases, you will generate a token that you can use to authenticate to the GraphQL API. For more information, see "Registering a GitHub App" and "About authentication with a GitHub App."

Authenticating with a OAuth app

To authenticate with an OAuth token from an OAuth app, you must first authorize your OAuth app using either a web application flow or device flow. Then, you can use the

access token that you received to access the API. For more information, see "Creating an OAuth app" and " Authorizing OAuth apps."

The GraphQL endpoint

The REST API has numerous endpoints; the GraphQL API has a single endpoint:

```
https://api.github.com/graphql
```

The endpoint remains constant no matter what operation you perform.

Communicating with GraphQL

Because GraphQL operations consist of multiline JSON, GitHub recommends using the Explorer to make GraphQL calls. You can also use `curl` or any other HTTP-speaking library.

In REST, HTTP verbs determine the operation performed. In GraphQL, you'll provide a JSON-encoded body whether you're performing a query or a mutation, so the HTTP verb is `POST`. The exception is an introspection query, which is a simple `GET` to the endpoint. For more information on GraphQL versus REST, see "Migrating from REST to GraphQL."

To query GraphQL in a `curl` command, make a `POST` request with a JSON payload. The payload must contain a string called `query`:

```
curl -H "Authorization: bearer TOKEN" -X POST -d " \
 { \
   \"query\": \"query { viewer { login }}\" \
 } \
" https://api.github.com/graphql
```

Note: The string value of `"query"` must escape newline characters or the schema will not parse it correctly. For the `POST` body, use outer double quotes and escaped inner double quotes.

About query and mutation operations

The two types of allowed operations in GitHub's GraphQL API are *queries* and *mutations*. Comparing GraphQL to REST, queries operate like `GET` requests, while mutations operate like `POST` / `PATCH` / `DELETE`. The mutation name determines which modification is executed.

For information about rate limiting, see "Resource limitations."

Queries and mutations share similar forms, with some important differences.

About queries

GraphQL queries return only the data you specify. To form a query, you must specify fields within fields (also known as *nested subfields*) until you return only scalars.

Queries are structured like this:

```
query {
  JSON-OBJECT-TO-RETURN
}
```

For a real-world example, see "Example query."

About mutations

To form a mutation, you must specify three things:

1. *Mutation name*. The type of modification you want to perform.
2. *Input object*. The data you want to send to the server, composed of *input fields*. Pass it as an argument to the mutation name.
3. *Payload object*. The data you want to return from the server, composed of *return fields*. Pass it as the body of the mutation name.

Mutations are structured like this:

```
mutation {
  MUTATION-NAME(input: {MUTATION-NAME-INPUT!}) {
    MUTATION-NAME-PAYLOAD
  }
}
```

The input object in this example is `MutationNameInput`, and the payload object is `MutationNamePayload`.

In the mutations reference, the listed *input fields* are what you pass as the input object. The listed *return fields* are what you pass as the payload object.

For a real-world example, see "Example mutation."

Working with variables

Variables can make queries more dynamic and powerful, and they can reduce complexity when passing mutation input objects.

> **Note**: If you're using the Explorer, make sure to enter variables in the separate Query Variables pane, and do not include the word `variables` before the JSON object.

Here's an example query with a single variable:

```
query($number_of_repos:Int!) {
  viewer {
    name
     repositories(last: $number_of_repos) {
       nodes {
         name
       }
     }
   }
}
variables {
   "number_of_repos": 3
}
```

There are three steps to using variables:

1. Define the variable outside the operation in a `variables` object:

   ```
   variables {
   ```

```
    "number_of_repos": 3
}
```

The object must be valid JSON. This example shows a simple `Int` variable type, but it's possible to define more complex variable types, such as input objects. You can also define multiple variables here.

2. Pass the variable to the operation as an argument:

```
query($number_of_repos:Int!){
```

The argument is a key-value pair, where the key is the *name* starting with `$` (e.g., `$number_of_repos`), and the value is the *type* (e.g., `Int`). Add a `!` to indicate whether the type is required. If you've defined multiple variables, include them here as multiple arguments.

3. Use the variable within the operation:

```
repositories(last: $number_of_repos) {
```

In this example, we substitute the variable for the number of repositories to retrieve. We specify a type in step 2 because GraphQL enforces strong typing.

This process makes the query argument dynamic. We can now simply change the value in the `variables` object and keep the rest of the query the same.

Using variables as arguments lets you dynamically update values in the `variables` object without changing the query.

Example query

Let's walk through a more complex query and put this information in context.

The following query looks up the `octocat/Hello-World` repository, finds the 20 most recent closed issues, and returns each issue's title, URL, and first 5 labels:

```
query {
  repository(owner:"octocat", name:"Hello-World") {
    issues(last:20, states:CLOSED) {
      edges {
        node {
          title
          url
          labels(first:5) {
            edges {
              node {
                name
              }
            }
          }
        }
      }
    }
  }
}
```

Looking at the composition line by line:

- `query {`

  Because we want to read data from the server, not modify it, `query` is the root

operation. (If you don't specify an operation, `query` is also the default.)

- `repository(owner:"octocat", name:"Hello-World") {`

  To begin the query, we want to find a `repository` object. The schema validation indicates this object requires an `owner` and a `name` argument.

- `issues(last:20, states:CLOSED) {`

  To account for all issues in the repository, we call the `issues` object. (We *could* query a single `issue` on a `repository`, but that would require us to know the number of the issue we want to return and provide it as an argument.)

  Some details about the `issues` object:

  - The docs tell us this object has the type `IssueConnection`.
  - Schema validation indicates this object requires a `last` or `first` number of results as an argument, so we provide `20`.
  - The docs also tell us this object accepts a `states` argument, which is an `IssueState` enum that accepts `OPEN` or `CLOSED` values. To find only closed issues, we give the `states` key a value of `CLOSED`.

- `edges {`

  We know `issues` is a connection because it has the `IssueConnection` type. To retrieve data about individual issues, we have to access the node via `edges`.

- `node {`

  Here we retrieve the node at the end of the edge. The `IssueConnection` docs indicate the node at the end of the `IssueConnection` type is an `Issue` object.

- Now that we know we're retrieving an `Issue` object, we can look at the docs and specify the fields we want to return:

  ```
  title
  url
  labels(first:5) {
    edges {
      node {
        name
      }
    }
  }
  ```

  Here we specify the `title`, `url`, and `labels` fields of the `Issue` object.

  The `labels` field has the type `LabelConnection`. As with the `issues` object, because `labels` is a connection, we must travel its edges to a connected node: the `label` object. At the node, we can specify the `label` object fields we want to return, in this case, `name`.

You may notice that running this query on the Octocat's public `Hello-World` repository won't return many labels. Try running it on one of your own repositories that does use labels, and you'll likely see a difference.

Example mutation

Mutations often require information that you can only find out by performing a query first. This example shows two operations:

1. A query to get an issue ID.

2. A mutation to add an emoji reaction to the issue.

```
query FindIssueID {
  repository(owner:"octocat", name:"Hello-World") {
    issue(number:349) {
      id
    }
  }
}

mutation AddReactionToIssue {
  addReaction(input:{subjectId:"MDU6SXNzdWUyMzEzOTE1NTE=",content:HOORAY}) {
    reaction {
      content
    }
    subject {
      id
    }
  }
}
```

> Although you can include a query and a mutation in the same Explorer window if you give them names (`FindIssueID` and `AddReactionToIssue` in this example), the operations will be executed as separate calls to the GraphQL endpoint. It's not possible to perform a query at the same time as a mutation, or vice versa.

Let's walk through the example. The task sounds simple: add an emoji reaction to an issue.

So how do we know to begin with a query? We don't, yet.

Because we want to modify data on the server (attach an emoji to an issue), we begin by searching the schema for a helpful mutation. The reference docs show the `addReaction` mutation, with this description: `Adds a reaction to a subject.` Perfect!

The docs for the mutation list three input fields:

- `clientMutationId` (`String`)
- `subjectId` (`ID!`)
- `content` (`ReactionContent!`)

The `!`s indicate that `subjectId` and `content` are required fields. A required `content` makes sense: we want to add a reaction, so we'll need to specify which emoji to use.

But why is `subjectId` required? It's because the `subjectId` is the only way to identify *which* issue in *which* repository to react to.

This is why we start this example with a query: to get the `ID`.

Let's examine the query line by line:

- `query FindIssueID {`

  Here we're performing a query, and we name it `FindIssueID`. Note that naming a query is optional; we give it a name here so that we can include it in same Explorer window as the mutation.

- `repository(owner:"octocat", name:"Hello-World") {`

  We specify the repository by querying the `repository` object and passing `owner` and `name` arguments.

- `issue(number:349) {`

We specify the issue to react to by querying the `issue` object and passing a `number` argument.

- `id`

  This is where we retrieve the `id` of `https://github.com/octocat/Hello-World/issues/349` to pass as the `subjectId`.

When we run the query, we get the `id`: `MDU6SXNzdWUyMzEzOTE1NTE=`

> **Note**: The `id` returned in the query is the value we'll pass as the `subjectID` in the mutation. Neither the docs nor schema introspection will indicate this relationship; you'll need to understand the concepts behind the names to figure this out.

With the ID known, we can proceed with the mutation:

- `mutation AddReactionToIssue {`

  Here we're performing a mutation, and we name it `AddReactionToIssue`. As with queries, naming a mutation is optional; we give it a name here so we can include it in the same Explorer window as the query.

- `addReaction(input:{subjectId:"MDU6SXNzdWUyMzEzOTE1NTE=",content:HOORAY}) {`

  Let's examine this line:

  - `addReaction` is the name of the mutation.
  - `input` is the required argument key. This will always be `input` for a mutation.
  - `{subjectId:"MDU6SXNzdWUyMzEzOTE1NTE=",content:HOORAY}` is the required argument value. This will always be an input object (hence the curly braces) composed of input fields (`subjectId` and `content` in this case) for a mutation.

  How do we know which value to use for the content? The `addReaction` docs tell us the `content` field has the type `ReactionContent`, which is an enum because only certain emoji reactions are supported on GitHub issues. These are the allowed values for reactions (note some values differ from their corresponding emoji names):

Content	Emoji
`+1`	
`-1`	
`laugh`	😄
`confused`	😕
`heart`	❤
`hooray`	
`rocket`	
`eyes`	

- The rest of the call is composed of the payload object. This is where we specify the data we want the server to return after we've performed the mutation. These lines come from the `addReaction` docs, which three possible return fields:

  - `clientMutationId` (`String`)

- `reaction` (`Reaction!`)
- `subject` (`Reactable!`)

In this example, we return the two required fields (`reaction` and `subject`), both of which have required subfields (respectively, `content` and `id`).

When we run the mutation, this is the response:

```
{
  "data": {
    "addReaction": {
      "reaction": {
        "content": "HOORAY"
      },
      "subject": {
        "id": "MDU6SXNzdWUyMTc5NTQ0OTc="
      }
    }
  }
}
```

That's it! Check out your reaction to the issue by hovering over the to find your username.

One final note: when you pass multiple fields in an input object, the syntax can get unwieldy. Moving the fields into a variable can help. Here's how you could rewrite the original mutation using a variable:

```
mutation($myVar:AddReactionInput!) {
  addReaction(input:$myVar) {
    reaction {
      content
    }
    subject {
      id
    }
  }
}
variables {
  "myVar": {
    "subjectId":"MDU6SXNzdWUyMTc5NTQ0OTc=",
    "content":"HOORAY"
  }
}
```

You may notice that the `content` field value in the earlier example (where it's used directly in the mutation) does not have quotes around `HOORAY`, but it does have quotes when used in the variable. There's a reason for this:

- When you use `content` directly in the mutation, the schema expects the value to be of type `ReactionContent`, which is an *enum*, not a string. Schema validation will throw an error if you add quotes around the enum value, as quotes are reserved for strings.
- When you use `content` in a variable, the variables section must be valid JSON, so the quotes are required. Schema validation correctly interprets the `ReactionContent` type when the variable is passed into the mutation during execution.

For more information on the difference between enums and strings, see the official GraphQL spec.

Further reading

There is a *lot* more you can do when forming GraphQL calls. Here are some places to

look next:

- Pagination
- Fragments
- Inline fragments
- Directives