Migrating from REST to GraphQL

Learn best practices and considerations for migrating from GitHub's REST API to GitHub's GraphQL API.

In this article

Differences in API logic

Migrating from REST to GraphQL represents a significant shift in API logic. The differences between REST as a style and GraphQL as a specification make it difficult—and often undesirable—to replace REST API calls with GraphQL API queries on a one-to-one basis. We've included specific examples of migration below.

To migrate your code from the REST API to the GraphQL API:

- Review the GraphQL spec
- Review GitHub's GraphQL schema
- Consider how any existing code you have currently interacts with the GitHub REST API
- Use Global Node IDs to reference objects between API versions

Significant advantages of GraphQL include:

- Getting the data you need and nothing more
- Nested fields
- Strong typing

Here are examples of each.

Example: Getting the data you need and nothing more

A single REST API call retrieves a list of your organization's members:

```
curl -v https://api.github.com/orgs/:org/members
```

The REST payload contains excessive data if your goal is to retrieve only member names and links to avatars. However, a GraphQL query returns only what you specify:

```
query {
    organization(login:"github") {
    membersWithRole(first: 100) {
      edges {
        node {
          name
          avatarUrl
        }
      }
```

```
    }
  }
}
```

Consider another example: retrieving a list of pull requests and checking if each one is mergeable. A call to the REST API retrieves a list of pull requests and their summary representations:

```
curl -v https://api.github.com/repos/:owner/:repo/pulls
```

Determining if a pull request is mergeable requires retrieving each pull request individually for its detailed representation (a large payload) and checking whether its `mergeable` attribute is true or false:

```
curl -v https://api.github.com/repos/:owner/:repo/pulls/:number
```

With GraphQL, you could retrieve only the `number` and `mergeable` attributes for each pull request:

```
query {
    repository(owner:"octocat", name:"Hello-World") {
    pullRequests(last: 10) {
      edges {
        node {
          number
          mergeable
        }
      }
    }
  }
}
```

Example: Nesting

Querying with nested fields lets you replace multiple REST calls with fewer GraphQL queries. For example, retrieving a pull request along with its commits, non-review comments, and reviews using the **REST API** requires four separate calls:

```
curl -v https://api.github.com/repos/:owner/:repo/pulls/:number
curl -v https://api.github.com/repos/:owner/:repo/pulls/:number/commits
curl -v https://api.github.com/repos/:owner/:repo/issues/:number/comments
curl -v https://api.github.com/repos/:owner/:repo/pulls/:number/reviews
```

Using the **GraphQL API**, you can retrieve the data with a single query using nested fields:

```
{
  repository(owner: "octocat", name: "Hello-World") {
    pullRequest(number: 1) {
      commits(first: 10) {
        edges {
          node {
            commit {
              oid
              message
            }
          }
        }
      }
      comments(first: 10) {
        edges {
          node {
```

```
            body
            author {
              login
            }
          }
        }
      }
      reviews(first: 10) {
        edges {
          node {
            state
          }
        }
      }
    }
  }
}
```

You can also extend the power of this query by substituting a variable for the pull request number.

Example: Strong typing

GraphQL schemas are strongly typed, making data handling safer.

Consider an example of adding a comment to an issue or pull request using a GraphQL mutation, and mistakenly specifying an integer rather than a string for the value of `clientMutationId`:

```
mutation {
  addComment(input:{clientMutationId: 1234, subjectId: "MDA6SXNzdWUyMjcyMDA2MTT=", body: "Looks good to me!"}) {
    clientMutationId
    commentEdge {
      node {
        body
        repository {
          id
          name
          nameWithOwner
        }
        issue {
          number
        }
      }
    }
  }
}
```

Executing this query returns errors specifying the expected types for the operation:

```
{
  "data": null,
  "errors": [
    {
      "message": "Argument 'input' on Field 'addComment' has an invalid value. Expected type 'AddCommentInput!'.",
      "locations": [
        {
          "line": 3,
          "column": 3
        }
      ]
    },
    {
      "message": "Argument 'clientMutationId' on InputObject 'AddCommentInput' has an invalid value. Expected type 'String'.",
```

```
      "locations": [
        {
          "line": 3,
          "column": 20
        }
      ]
    }
  ]
}
```

Wrapping `1234` in quotes transforms the value from an integer into a string, the expected type:

```
mutation {
  addComment(input:{clientMutationId: "1234", subjectId: "MDA6SXNzdWUyMjcyMDA2MTT=", body: "Looks good to me!"}) {
    clientMutationId
    commentEdge {
      node {
        body
        repository {
          id
          name
          nameWithOwner
        }
        issue {
          number
        }
      }
    }
  }
}
```