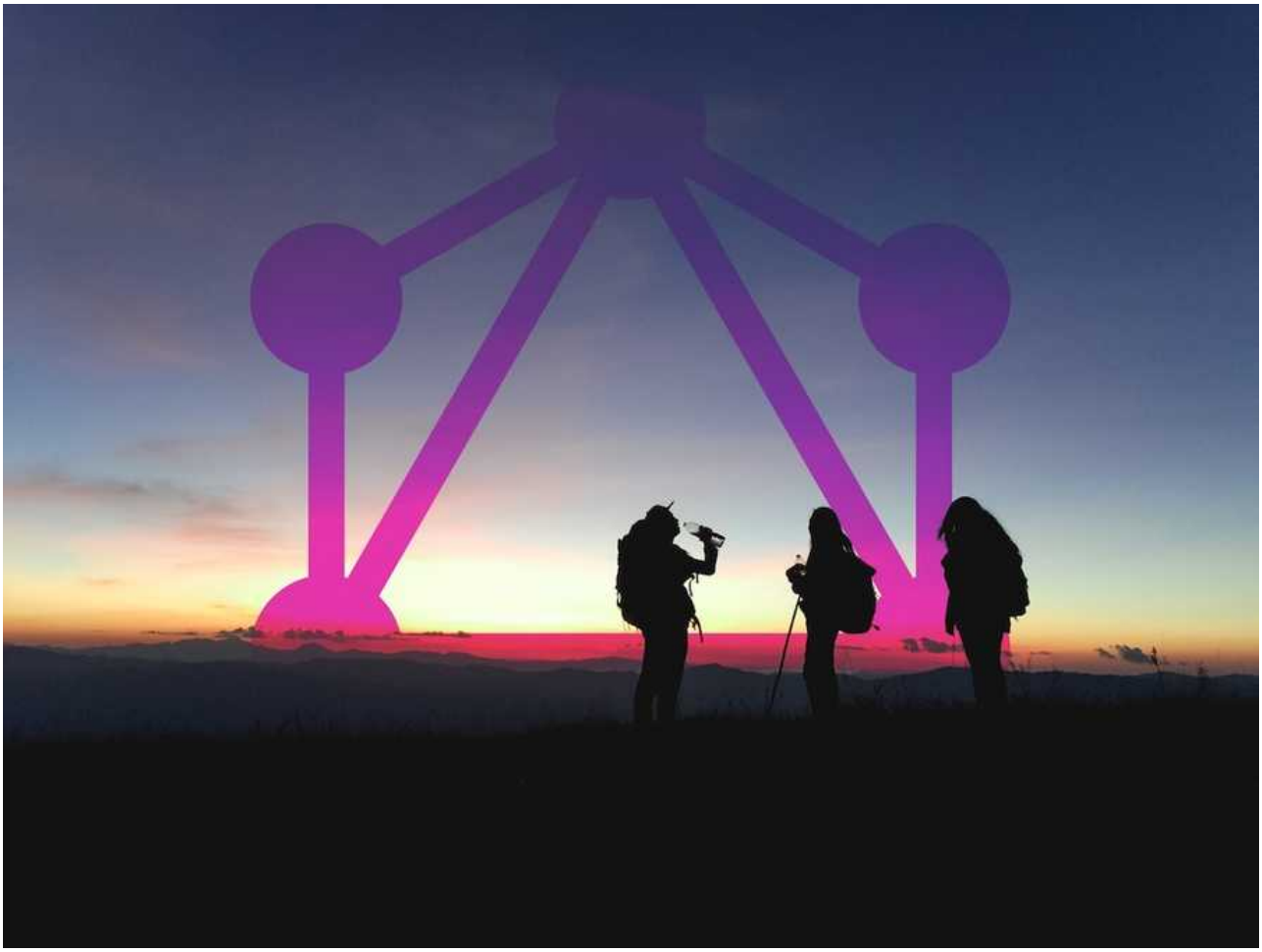
GraphQL Tutorial for Beginners

NOVEMBER 01, 2018 BY ROBIN WIERUCH - EDIT THIS POST



Interested in reading this tutorial as one of many chapters in my GraphQL book? Checkout the entire The Road to GraphQL book that teaches you to become a fullstack developer with JavaScript.

This tutorial is part 2 of 5 in this series.

Part 1: Getting Started with GitHub's GraphQL API
Part 3: A complete React with GraphQL Tutorial
Part 4: Apollo Client Tutorial for Beginners
Part 5: React with Apollo and GraphQL Tutorial

Before we start to build full-fledged GraphQL applications, on the client- and server-side, let's explore GraphQL with the tools we have installed in the previous sections. You can either use GraphiQL or the GitHub's GraphQL Explorer. In the following, you will learn about GraphQL's fundamentals by executing your first GraphQL queries, mutations and even by exploring features such as pagination, in the context of GitHub's GraphQL API.

GRAPHQL QUERY WITH GITHUB'S GRAPHQL API

In this section, you will interact with the GitHub API using queries and mutations without React, so you can use your GraphiQL application or GitHub's GraphQL Explorer to make GraphQL query requests to GitHub's API. Both tools should be authorized to make requests using a personal access token. On the left-hand side of your GraphiQL application, you can fill in GraphQL queries and mutations. Add the following query to request data about yourself.

```
{
  viewer {
    name
    url
  }
}
```

The `viewer` object can be used to request data about the currently authorized user. Since you are authorized by your personal access token, it should show data about your account. The `viewer` is an **object** in GraphQL terms. Objects hold data about an entity. This data is accessed using a so-called **field** in GraphQL. Fields are used to ask for specific properties in objects. For instance, the `viewer` object exposes a wide range of fields. Two fields for the object--`name` and `url`--were used in the query. In its most basic form, a query is just objects and fields, and objects can also be called fields.

Once you run the query in GraphiQL, you should see output similar to the one below, where

Once you run the query in GraphiQL, you should see output similar to the one below, where your name and URL are in the place of mine:

```
{
  "data": {
    "viewer": {
      "name": "Robin Wieruch",
      "url": "https://github.com/rwieruch"
    }
  }
}
```

Congratulations, you have performed your first query to access fields from your own user data. Now, let's see how to request data from a source other than yourself, like a public GitHub organization. To specify a GitHub organization, you can pass an **argument** to fields:

```
{
  organization(login: "the-road-to-learn-react") {
    name
    url
  }
}
```

When using GitHub's API, an organization is identified with a `login`. If you have used GitHub before, you might know this is a part of the organization URL:

```
https://github.com/the-road-to-learn-react
```

By providing a `login` to identify the organization, you can request data about it. In this example, you have specified two fields to access data about the organization's `name` and `url`. The request should return something similar to the following output:

```
{
  "data": {
    "organization": {
      "name": "The Road to learn React",
      "url": "https://github.com/the-road-to-learn-react"
    }
  }
}
```

In the previous query you passed an argument to a field. As you can imagine, you can add arguments to various fields using GraphQL. It grants a great deal of flexibility for structuring

arguments to various fields using GraphQL. It grants a great deal of flexibility for structuring queries, because you can make specifications to requests on a field level. Also, arguments can be of different types. With the organization above, you provided an argument with the type `String`, though you can also pass types like enumerations with a fixed set of options, integers, or booleans.

If you ever wanted to request data about two identical objects, you would have to use **aliases** in GraphQL. The following query wouldn't be possible, because GraphQL wouldn't know how to resolve the two organization objects in a result:

```
{
  organization(login: "the-road-to-learn-react") {
    name
    url
  }
  organization(login: "facebook") {
    name
    url
  }
}
```

You'd see an error such as `Field 'organization' has an argument conflict`. Using aliases, you can resolve the result into two blocks:

```
{
  book: organization(login: "the-road-to-learn-react") {
    name
    url
  }
  company: organization(login: "facebook") {
    name
    url
  }
}
```

The result should be similar to the following:

```
{
  "data": {
    "book": {
      "name": "The Road to learn React",
      "url": "https://github.com/the-road-to-learn-react"
    },
    "company": {
      "name": "Facebook",
```

```
        "url": "https://github.com/facebook"
      }

    }
  }
```

Next, imagine you want to request multiple fields for both organizations. Re-typing all the fields for each organization would make the query repetitive and verbose, so we'll use **fragments** to extract the query's reusable parts. Fragments are especially useful when your query becomes deeply nested and uses lots of shared fields.

```
{
  book: organization(login: "the-road-to-learn-react") {
    ...sharedOrganizationFields
  }
  company: organization(login: "facebook") {
    ...sharedOrganizationFields
  }
}

fragment sharedOrganizationFields on Organization {
  name
  url
}
```

As you can see, you have to specify on which **type** of object the fragment should be used. In this case, it is the type `Organization`, which is a custom type defined by GitHub's GraphQL API. This is how you use fragments to extract and reuse parts of your queries. At this point, you might want to open "Docs" on the right side of your GraphiQL application. The documentation gives you access to the GraphQL **schema**. A schema exposes the GraphQL API used by your GraphiQL application, which is Github's GraphQL API in this case. It defines the GraphQL **graph** that is accessible via the GraphQL API using queries and mutations. Since it is a graph, objects and fields can be deeply nested in it, which we'll certainly encounter as we move along.

Since we're exploring queries and not mutations at the moment, select "Query" in the "Docs" sidebar. Afterward, traverse the objects and fields of the graph, explore their optional arguments. By clicking them, you can see the accessible fields within those objects in the graph. Some fields are common GraphQL types such as `String`, `Int` and `Boolean`, while some other types are **custom types** like the `Organization` type we used. In addition, you can see whether arguments are required when requesting fields on an object. It can be identified by the exclamation point. For instance, a field with a `String!` argument requires that you pass in a `String` argument whereas a field with a `String` argument doesn't require

you to pass it.

In the previous queries, you provided arguments that identified an organization to your fields; but you **inlined these arguments** in your query. Think about a query like a function, where it's important to provide dynamic arguments to it. That's where the **variable** in GraphQL comes in, as it allows arguments to be extracted as variables from queries. Here's how an organization's `login` argument can be extracted to a dynamic variable:

```
query ($organization: String!) {
  organization(login: $organization) {
    name
    url
  }
}
```

It defines the `organization` argument as a variable using the `$` sign. Also, the argument's type is defined as a `String`. Since the argument is required to fulfil the query, the `String` type has an exclamation point.

In the "Query Variables" panel, the variables would have the following content for providing the `organization` variable as argument for the query:

```
{
  "organization": "the-road-to-learn-react"
}
```

Essentially, variables can be used to create dynamic queries. Following the best practices in GraphQL, we don't need manual string interpolation to structure a dynamic query later on. Instead, we provide a query that uses variables as arguments, which are available when the query is sent as a request to the GraphQL API. You will see both implementations later in your React application.

Sidenote: You can also define a **default variable** in GraphQL. It has to be a non-required argument, or an error will occur about a **nullable variable** or **non-null variable**. For learning about default variables, we'll make the `organization` argument non-required by omitting the exclamation point. Afterwards, it can be passed as a default variable.

```
query ($organization: String = "the-road-to-learn-react") {
  organization(login: $organization) {
    name
    url
```

```
  }
}
```

Try to execute the previous query with two sets of variables: once with the `organization` variable that's different from the default variable, and once without defining the `organization` variable.

Now, let's take a step back to examine the structure of the GraphQL query. After you introduced variables, you encountered the `query` statement in your query structure for the first time. Before, you used the **shorthand version of a query** by omitting the `query` statement, but the `query` statement has to be there now that it's using variables. Try the following query without variables, but with the `query` statement, to verify that the long version of a query works.

```
query {
  organization(login: "the-road-to-learn-react") {
    name
    url
  }
}
```



While it's not the shorthand version of the query, it still returns the same data as before, which is the desired outcome. The query statement is also called **operation type** in GraphQL lingua. For instance, it can also be a `mutation` statement. In addition to the operation type, you can also define an **operation name**.

```
query OrganizationForLearningReact {
  organization(login: "the-road-to-learn-react") {
    name
    url
  }
}
```

Compare it to anonymous and named functions in your code. A **named query** provides a certain level of clarity about what you want to achieve with the query in a declarative way, and it helps with debugging multiple queries, so it should be used when you want to implement an application. Your final query, without showing the variables panel again, could look like the following:

```
query OrganizationForLearningReact($organization: String!) {
  organization(login: $organization) {
```

```
    name
    url
  }
}
```

So far you've only accessed one object, an organization with a couple of its fields. The GraphQL schema implements a whole graph, so let's see how to access a **nested object** from within the graph with a query. It's not much different from before:

```
query OrganizationForLearningReact(
  $organization: String!,
  $repository: String!
) {
  organization(login: $organization) {
    name
    url
    repository(name: $repository) {
      name
    }
  }
}
```

Provide a second variable to request a specific repository of the organization:

```
{
  "organization": "the-road-to-learn-react",
  "repository": "the-road-to-learn-react-chinese"
}
```

The organization that teaches about React has translated versions of its content, and one of its repositories teaches students about React in simplified Chinese. Fields in GraphQL can be nested objects again, and you have queried two associated objects from the graph. The requests are made on a graph that can have a deeply nested structure. While exploring the "Docs" sidebar in GraphiQL before, you might have seen that you can jump from object to object in the graph.

A **directive** can be used to query data from your GraphQL API in a more powerful way, and they can be applied to fields and objects. Below, we use two types of directives: an **include directive**, which includes the field when the Boolean type is set to true; and the **skip directive**, which excludes it instead. With these directives, you can apply conditional structures to your shape of query. The following query showcases the include directive, but you can substitute it with the skip directive to achieve the opposite effect:

```
```

```
query OrganizationForLearningReact(
  $organization: String!,
  $repository: String!,
  $withFork: Boolean!
) {
  organization(login: $organization) {
    name
    url
    repository(name: $repository) {
      name
      forkCount @include(if: $withFork)
    }
  }
}
```

Now you can decide whether to include the information for the `forkCount` field based on provided variables.

```
{
  "organization": "the-road-to-learn-react",
  "repository": "the-road-to-learn-react-chinese",
  "withFork": true
}
```

The query in GraphQL gives you all you need to read data from a GraphQL API. The last section may have felt like a whirlwind of information, so these exercises provide additional practice until you feel comfortable.

Exercises:

- Read more about the Query in GraphQL.
- Explore GitHub's query schema by using the "Docs" sidebar in GraphiQL.
- Create several queries to request data from GitHub's GraphQL API using the following features:
  - objects and fields
  - nested objects
  - fragments
  - arguments and variables
  - operation names
  - directives

GRAPHQL MUTATION WITH GITHUB'S

GRAPHQL MUTATION WITH GITHUB'S GRAPHQL API

This section introduces the GraphQL mutation. It complements the GraphQL query because it is used for writing data instead of reading it. The mutation shares the same principles as the query: it has fields and objects, arguments and variables, fragments and operation names, as well as directives and nested objects for the returned result. With mutations you can specify data as fields and objects that should be returned after it 'mutates' into something acceptable. Before you start making your first mutation, be aware that you are using live GitHub data, so if you follow a person on GitHub using your experimental mutation, you will follow this person for real. Fortunately this sort of behavior is encouraged on GitHub.

In this section, you will star a repository on GitHub, the same one you used a query to request before, using a mutation from GitHub's API. You can find the `addStar` mutation in the "Docs" sidebar. The repository is a project for teaching developers about the fundamentals of React, so starring it should prove useful.



You can visit the repository to see if you've given a star to the repository already. We want an unstarred repository so we can star it using a mutation. Before you can star a repository, you need to know its identifier, which can be retrieved by a query:

```
query {
  organization(login: "the-road-to-learn-react") {
    name
    url
    repository(name: "the-road-to-learn-react") {
      id
      name
    }
  }
}
```

In the results for the query in GraphiQL, you should see the identifier for the repository:

```
MDEwOlJlcG9zaXRvcnk2MzM1MjkwNw==
```

Before using the identifier as a variable, you can structure your mutation in GraphiQL the following way:

```
mutation AddStar($repositoryId: ID!) {
  addStar(input: { starrableId: $repositoryId }) {
    starrable {
```

```
      starrable {
        id
        viewerHasStarred
      }
    }
  }
```

The mutation's name is given by GitHub's API: `addStar`. You are required to pass it the `starrableId` as `input` to identify the repository; otherwise, the GitHub server won't know which repository to star with the mutation. In addition, the mutation is a named mutation: `AddStar`. It's up to you to give it any name. Last but not least, you can define the return values of the mutation by using objects and fields again. It's identical to a query. Finally, the variables tab provides the variable for the mutation you retrieved with the last query:

```
{
  "repositoryId": "MDEwOlJlcG9zaXRvcnk2MzM1MjkwNw=="
}
```

Once you execute the mutation, the result should look like the following. Since you specified the return values of your mutation using the `id` and `viewerHasStarred` fields, you should see them in the result.

```
{
  "data": {
    "addStar": {
      "starrable": {
        "id": "MDEwOlJlcG9zaXRvcnk2MzM1MjkwNw==",
        "viewerHasStarred": true
      }
    }
  }
}
```

The repository is starred now. It's visible in the result, but you can verify it in the repository on GitHub. Congratulations, you made your first mutation.

Exercises:

- Read more about the Mutation in GraphQL
- Explore GitHub's mutations by using the "Docs" sidebar in GraphiQL
- Find GitHub's `addStar` mutation in the "Docs" sidebar in GraphiQL
  - Check its possible fields for returning a response
- Create a few other mutations for this or another repository such as:

- Create a few other mutations for this or another repository such as:
  - Unstar repository
  - Watch repository
- Create two named mutations side by side in the GraphiQL panel and execute them
- Read more about the schema and types
  - Make yourself a picture of it, but don't worry if you don't understand everything yet

---

GRAPHQL PAGINATION

This is where we return to the concept of **pagination** mentioned in the first chapter. Imagine you have a list of repositories in your GitHub organization, but you only want to retrieve a few of them to display in your UI. It could take ages to fetch a list of repositories from a large organization. In GraphQL, you can request paginated data by providing arguments to a **list field**, such as an argument that says how many items you are expecting from the list.

```
query OrganizationForLearningReact {
  organization(login: "the-road-to-learn-react") {
    name
    url
    repositories(first: 2) {
      edges {
        node {
          name
        }
      }
    }
  }
}
```

A `first` argument is passed to the `repositories` list field that specifies how many items from the list are expected in the result. The query shape doesn't need to follow the `edges` and `node` structure, but it's one of a few solutions to define paginated data structures and lists with GraphQL. Actually, it follows the interface description of Facebook's GraphQL client called Relay. GitHub followed this approach and adopted it for their own GraphQL pagination API. Later, you will learn in the exercises about other strategies to implement pagination with GraphQL.

After executing the query, you should see two items from the list in the repositories field. We still need to figure out how to fetch the next two repositories in the list, however. The first result of the query is the first **page** of the paginated list, the second query result should be

the second page. In the following, you will see how the query structure for paginated data

allows us to retrieve meta information to execute successive queries. For instance, each edge comes with its own cursor field to identify its position in the list.

```
query OrganizationForLearningReact {
  organization(login: "the-road-to-learn-react") {
    name
    url
    repositories(first: 2) {
      edges {
        node {
          name
        }
        cursor
      }
    }
  }
}
```



The result should be similar to the following:

```
{
  "data": {
    "organization": {
      "name": "The Road to learn React",
      "url": "https://github.com/the-road-to-learn-react",
      "repositories": {
        "edges": [
          {
            "node": {
              "name": "the-road-to-learn-react"
            },
            "cursor": "Y3Vyc29yOnYyOpHOA8awSw=="
          },
          {
            "node": {
              "name": "hackernews-client"
            },
            "cursor": "Y3Vyc29yOnYyOpHOBGhimw=="
          }
        ]
      }
    }
  }
}
```

Now, you can use the cursor of the first repository in the list to execute a second query. By

using the `after` argument for the `repositories` list field, you can specify an entry point to

retrieve your next page of paginated data. What would the result look like when executing the following query?

```
query OrganizationForLearningReact {
  organization(login: "the-road-to-learn-react") {
    name
    url
    repositories(first: 2, after: "Y3Vyc29yOnYyOpHOA8awSw==") {
      edges {
        node {
          name
        }
        cursor
      }
    }
  }
}
```

In the previous result, only the second item is retrieved, as well as a new third item. The first item isn't retrieved because you have used its cursor as `after` argument to retrieve all items after it. Now you can imagine how to make successive queries for paginated lists:

- Execute the initial query without a cursor argument
- Execute every following query with the cursor of the **last** item's cursor from the previous query result

To keep the query dynamic, we extract its arguments as variables. Afterward, you can use the query with a dynamic `cursor` argument by providing a variable for it. The `after` argument can be `undefined` to retrieve the first page. In conclusion, that would be everything you need to fetch pages of lists from one large list by using a feature called pagination. You need a mandatory argument specifying how many items should be retrieved and an optional argument, in this case the `after` argument, specifying the starting point for the list.

There are also a couple helpful ways to use meta information for your paginated list. Retrieving the `cursor` field for every repository may be verbose when using only the `cursor` of the last repository, so you can remove the `cursor` field for an individual edge, but add the `pageInfo` object with its `endCursor` and `hasNextPage` fields. You can also request the `totalCount` of the list.

```
query OrganizationForLearningReact {
  organization(login: "the-road-to-learn-react") {
```

```
  organization(login: "the-road-to-learn-react") {
    name
    url

    repositories(first: 2, after: "Y3Vyc29yOnYyOpHOA8awSw==") {
      totalCount
      edges {
        node {
          name
        }
      }
      pageInfo {
        endCursor
        hasNextPage
      }
    }
  }
}
```

The `totalCount` field discloses the total number of items in the list, while the `pageInfo` field gives you information about two things:

- **`endCursor`** can be used to retrieve the successive list, which we did with the `cursor` field, except this time we only need one meta field to perform it. The cursor of the last list item is sufficient to request the next page of list.
- **`hasNextPage`** gives you information about whether or not there is a next page to retrieve from the GraphQL API. Sometimes you've already fetched the last page from your server. For applications that use infinite scrolling to load more pages when scrolling lists, you can stop fetching pages when there are no more available.

This meta information completes the pagination implementation. Information is made accessible using the GraphQL API to implement paginated lists and infinite scroll. Note, this covers GitHub's GraphQL API; a different GraphQL API for pagination might use different naming conventions for the fields, exclude meta information, or employ different mechanisms altogether.

Exercises:

- Extract the `login` and the `cursor` from your pagination query as variables.
- Exchange the `first` argument with a `last` argument.
- Search for the `repositories` field in the GraphiQL "Docs" sidebar which says: "A list of repositories that the ... owns."
  - Explore the other arguments that can be passed to this list field.
  - Use the `orderBy` argument to retrieve an ascending or descending list.

- Read more about pagination in GraphQL.
  - The cursor approach is only one solution which is used by GitHub.
  - Make sure to understand the other solutions, too.

. . .

Interacting with GitHub's GraphQL API via GraphiQL or GitHub's GraphQL Explorer is only the beginning. You should be familiar with the fundamental GraphQL concepts now. But there are a lot more exciting concepts to explore. In the next chapters, you will implement a fully working GraphQL client application with React that interacts with GitHub's API.

This tutorial is part 2 of 5 in this series.

Part 1: Getting Started with GitHub's GraphQL API
Part 3: A complete React with GraphQL Tutorial
Part 4: Apollo Client Tutorial for Beginners
Part 5: React with Apollo and GraphQL Tutorial

KEEP READING ABOUT GRAPHQL ›

A COMPLETE REACT WITH GRAPHQL TUTORIAL

In this client-sided GraphQL application we'll build together, you will learn how to combine React with GraphQL. There is no clever library like Apollo Client or Relay to help you get started yet...

REACT WITH APOLLO AND GRAPHQL TUTORIAL

In this tutorial, you will learn how to combine React with GraphQL in your application using Apollo. The Apollo toolset can be used to create a GraphQL client, GraphQL server, and other complementary...

THE ROAD TO REACT

Learn React by building real world applications. No setup configuration. No tooling. Plain React in 200+ pages of learning material. Learn React like **50.000+ readers**.

GET THE BOOK

Get it on Amazon.

TAKE PART

NEVER MISS AN ARTICLE ABOUT WEB DEVELOPMENT AND JAVASCRIPT.

✔ Join 50.000+ Developers

- ✔ Learn Web Development
- ✔ Learn JavaScript
- ✔ Access Tutorials, eBooks and Courses
- ✔ Personal Development as a Software Engineer

SUBSCRIBE >

View our Privacy Policy.



PORTFOLIO

Online Courses

Open Source

Tutorials

ABOUT

About me

What I use

How to work with me

How to support me