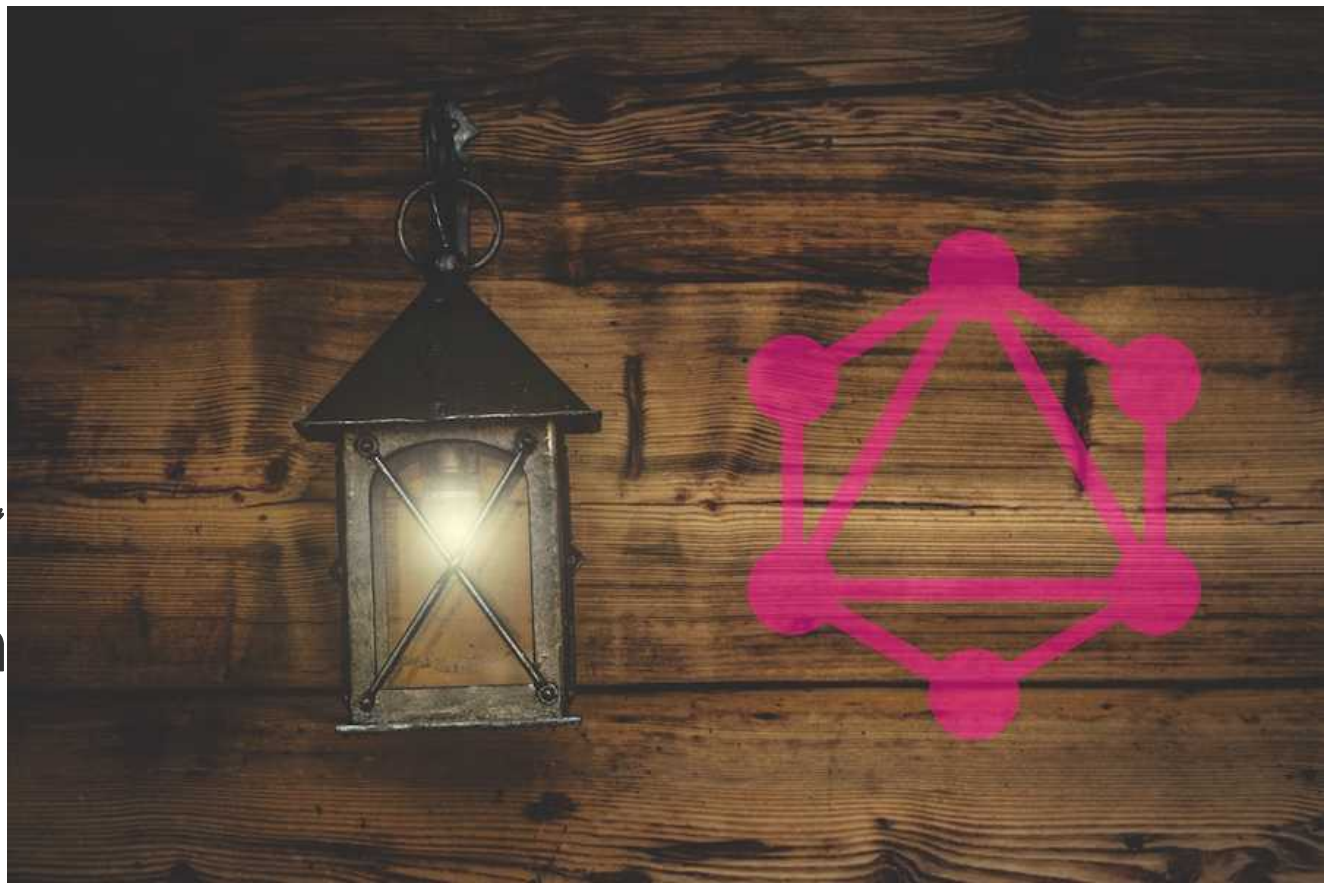
A complete React with GraphQL Tutorial

APRIL 09, 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 3 of 5 in this series.

Part 1: Getting Started with GitHub's GraphQL API
Part 2: GraphQL Tutorial for Beginners
Part 4: Apollo Client Tutorial for Beginners
Part 5: React with Apollo and 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, so instead, you will perform GraphQL queries and mutations with basic HTTP requests. Later, in the next application we are going to build together, I'll introduce Apollo as a GraphQL client for your React.js application. For now, the application we build should only show how to use GraphQL in React with HTTP.

Along the way, you will build a simplified GitHub client, basically an issue tracker for GitHub, that consumes GitHub's GraphQL API. You will perform GraphQL queries and mutations to read and write data, and by the end, you should be able to showcase a GraphQL in React example that can be used by other developers as a learning tool. The final application you are going to build can be found in this repository on GitHub.

TABLE OF CONTENTS

WRITING YOUR FIRST REACT GRAPHQL CLIENT

After the last sections, you should be ready to use queries and mutations in your React application. In this section, you will create a React application that consumes the GitHub GraphQL API. The application should show open issues in a GitHub repository, making it a simple issue tracker. Again, if you lack experience with React, see The Road to learn React to learn more about it. After that you should be well set up for the following section.

For this application, no elaborate React setup is needed. You will simply use create-react-app to create your React application with zero-configuration. If you want to have an elaborated React setup instead, read this setup guide for using Webpack with React. For now, let's create the application with create-react-app. In your general projects folder, type the following instructions:

```
npx create-react-app react-graphql-github-vanilla
cd react-graphql-github-vanilla
```

After your application has been created, you can test it with `npm start` and `npm test`. Again, after you have learned about plain React in *the Road to learn React*, you should be familiar with npm, create-react-app, and React itself.

The following application will focus on the *src/App.js* file. It's up to you to split out components, configuration, or functions to their own folders and files. Let's get started with the App component in the mentioned file. In order to simplify it, you can change it to the following content:

```
import React, { Component } from 'react';

const TITLE = 'React GraphQL GitHub Client';

class App extends Component {
  render() {
    return (
      <div>
        <h1>{TITLE}</h1>
      </div>
    );
  }
}

export default App;
```

The component only renders a `title` as a headline. Before implementing any more React components, let's install a library to handle GraphQL requests, executing queries and mutations, using a HTTP POST method. For this, you will use axios. On the command line, type the following command to install axios in the project folder:

```
npm install axios --save
```

Afterward, you can import axios next to your App component and configure it. It's perfect for the following application, because somehow you want to configure it only once with your personal

access token and GitHub's GraphQL API.

First, define a base URL for axios when creating a configured instance from it. As mentioned before, you don't need to define GitHub's URL endpoint every time you make a request because all queries and mutations point to the same URL endpoint in GraphQL. You get the flexibility from your query and mutation structures using objects and fields instead.

```
import React, { Component } from 'react';
import axios from 'axios';

const axiosGitHubGraphQL = axios.create({
  baseURL: 'https://api.github.com/graphql',
});

...

export default App;
```

Second, pass the personal access token as header to the configuration. The header is used by each request made with this axios instance.

```
...

const axiosGitHubGraphQL = axios.create({
  baseURL: 'https://api.github.com/graphql',
  headers: {
    Authorization: 'bearer YOUR_GITHUB_PERSONAL_ACCESS_TOKEN',
  },
});

...
```

Replace the `YOUR_GITHUB_PERSONAL_ACCESS_TOKEN` string with your personal access token. To avoid cutting and pasting your access token directly into the source code, you can create a *.env* file to hold all your environment variables on the command line in your project folder. If you don't want to share the personal token in a public GitHub repository, you can add the file to your *.gitignore*.

```
touch .env
```

Environment variables are defined in this *.env* file. Be sure to follow the correct naming constraints when using create-react-app, which uses `REACT_APP` as prefix for each key. In your

.env file, paste the following key value pair. The key has to have the `REACT_APP` prefix, and the value has to be your personal access token from GitHub.

```
REACT_APP_GITHUB_PERSONAL_ACCESS_TOKEN=xxxXXX
```

Now, you can pass the personal access token as environment variable to your axios configuration with string interpolation (template literals) to create a configured axios instance.

```
...

const axiosGitHubGraphQL = axios.create({
  baseURL: 'https://api.github.com/graphql',
  headers: {
    Authorization: `bearer ${
      process.env.REACT_APP_GITHUB_PERSONAL_ACCESS_TOKEN
    }`,
  },
});

...
```

The initial axios setup is essentially the same as we completed using the GraphiQL application before to access GitHub's GraphQL API, when you had to set a header with a personal access token and endpoint URL as well.

Next, set up a form for capturing details about a GitHub organization and repository from a user. It should be possible to fill out an input field to request a paginated list of issues for a specific GitHub repository. First, there needs to be a form with an input field to enter the organization and repository. The input field has to update React's local state. Second, the form needs a submit button to request data about the organization and repository that the user provided in the input field, which are located in the component's local state. Third, it would be convenient to have an initial local state for the organization and repository to request initial data when the component mounts for the first time.

Let's tackle implementing this scenario in two steps. The render method has to render a form with an input field. The form has to have an `onSubmit` handler, and the input field needs an `onChange` handler. The input field uses the `path` from the local state as a value to be a controlled component. The `path` value in the local state from the `onChange` handler updates in the second step.

```
class App extends Component {
  render() {
```

```
    return (
      <div>
        <h1>{TITLE}</h1>

        <form onSubmit={this.onSubmit}>
          <label htmlFor="url">
            Show open issues for https://github.com/
          </label>
          <input
            id="url"
            type="text"
            onChange={this.onChange}
            style={{ width: '300px' }}
          />
          <button type="submit">Search</button>
        </form>

        <hr />

        {/* Here comes the result! */}
      </div>
    );
  }
}
```

Declare the class methods to be used in the render method. The `componentDidMount()` lifecycle method can be used to make an initial request when the App component mounts. There needs to be an initial state for the input field to make an initial request in this lifecycle method.

```
class App extends Component {
  state = {
    path: 'the-road-to-learn-react/the-road-to-learn-react',
  };

  componentDidMount() {
    // fetch data
  }

  onChange = event => {
    this.setState({ path: event.target.value });
  };

  onSubmit = event => {
    // fetch data

    event.preventDefault();
  };

  render() {
    ...
  }
}
```

The previous implementation uses a React class component syntax you might have not used before. If you are not familiar with it, check this GitHub repository to gain more understanding. Using **class field declarations** lets you omit the constructor statement for initializing the local state, and eliminates the need to bind class methods. Instead, arrow functions will handle all the binding.

Following a best practice in React, make the input field a controlled component. The input element shouldn't be used to handle its internal state using native HTML behavior; it should be React.

```
class App extends Component {
  ...

  render() {
    const { path } = this.state;

    return (
      <div>
        <h1>{TITLE}</h1>

        <form onSubmit={this.onSubmit}>
          <label htmlFor="url">
            Show open issues for https://github.com/
          </label>
          <input
            id="url"
            type="text"
            value={path}
            onChange={this.onChange}
            style={{ width: '300px' }}
          />
          <button type="submit">Search</button>
        </form>

        <hr />

        {/* Here comes the result! */}
      </div>
    );
  }
}
```

The previous setup for the form--using input field(s), a submit button, `onChange()` and `onSubmit()` class methods--is a common way to implement forms in React. The only addition is the initial data fetching in the `componentDidMount()` lifecycle method to improve user

experience by providing an initial state for the query to request data from the backend. It is a useful foundation for fetching data from a third-party API in React.

When you start the application on the command line, you should see the initial state for the `path` in the input field. You should be able to change the state by entering something else in the input field, but nothing happens with `componentDidMount()` and submitting the form yet.

You might wonder why there is only one input field to grab the information about the organization and repository. When opening up a repository on GitHub, you can see that the organization and repository are encoded in the URL, so it becomes a convenient way to show the same URL pattern for the input field. You can also split the `organization/repository` later at the `/` to get these values and perform the GraphQL query request.

Exercises:

- Confirm your source code for the last section
  - Confirm the changes from the last section
- If you are unfamiliar with React, check out *The Road to learn React*

REACT GRAPHQL QUERY

In this section, you are going to implement your first GraphQL query in React, fetching issues from an organization's repository, though not all at once. Start by fetching only an organization. Let's define the query as a variable above of the App component.

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

Use template literals in JavaScript to define the query as string with multiple lines. It should be identical to the query you used before in GraphiQL or GitHub Explorer. Now, you can use axios to make a POST request to GitHub's GraphiQL API. The configuration for axios already points to the correct API endpoint and uses your personal access token. The only thing left is passing the query to it as payload during a POST request. The argument for the endpoint can be an empty string, because you defined the endpoint in the configuration. It will execute the request when the App

component mounts in `componentDidMount()`. After the promise from axios has been resolved, only a console log of the result remains.

```
...

const axiosGitHubGraphQL = axios.create({
  baseURL: 'https://api.github.com/graphql',
  headers: {
    Authorization: `bearer ${
      process.env.REACT_APP_GITHUB_PERSONAL_ACCESS_TOKEN
    }`,
  },
});

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

class App extends Component {
  ...

  componentDidMount() {
    this.onFetchFromGitHub();
  }

  onSubmit = event => {
    // fetch data

    event.preventDefault();
  };

  onFetchFromGitHub = () => {
    axiosGitHubGraphQL
      .post('', { query: GET_ORGANIZATION })
      .then(result => console.log(result));
  };

  ...
}
```



You used only axios to perform a HTTP POST request with a GraphQL query as payload. Since axios uses promises, the promise resolves eventually and you should have the result from the GraphQL API in your hands. There is nothing magical about it. It's an implementation in plain JavaScript using axios as HTTP client to perform the GraphQL request with plain HTTP.

Start your application again and verify that you have got the result in your developer console log. If you get a 401 HTTP status code, you didn't set up your personal access token properly. Otherwise, if everything went fine, you should see a similar result in your developer console log.

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

The top level information is everything axios returns you as meta information for the request. It's all axios, and nothing related to GraphQL yet, which is why most of it is substituted with a placeholder. Axios has a `data` property that shows the result of your axios request. Then again comes a `data` property which reflects the GraphQL result. At first, the `data` property seems redundant in the first result, but once you examine it you will know that one `data` property comes from axios, while the other comes from the GraphQL data structure. Finally, you find the result of the GraphQL query in the second `data` property. There, you should find the organization with its resolved name and url fields as string properties.

In the next step, you're going to store the result holding the information about the organization in React's local state. You will also store potential errors in the state if any occur.

```
class App extends Component {
  state = {
    path: 'the-road-to-learn-react/the-road-to-learn-react',
    organization: null,
    errors: null,
  };

  ...

  onFetchFromGitHub = () => {
    axiosGitHubGraphQL
      .post('', { query: GET_ORGANIZATION })
      .then(result =>
        this.setState(() => ({
          organization: result.data.data.organization,
```

```
          errors: result.data.errors,
        })),
      );
  }

  ...

}
```

In the second step, you can display the information about the organization in your App component's `render()` method:

```
class App extends Component {
  ...

  render() {
    const { path, organization } = this.state;

    return (
      <div>
        <h1>{TITLE}</h1>

        <form onSubmit={this.onSubmit}>
          ...
        </form>

        <hr />

        <Organization organization={organization} />
      </div>
    );
  }
}
```

Introduce the Organization component as a new functional stateless component to keep the render method of the App component concise. Because this application is going to be a simple GitHub issue tracker, you can already mention it in a short paragraph.

```
class App extends Component {
  ...
}

const Organization = ({ organization }) => (
  <div>
    <p>
      <strong>Issues from Organization:</strong>
      <a href={organization.url}>{organization.name}</a>
    </p>
  </div>
```

```
  );
```

In the final step, you have to decide what should be rendered when nothing is fetched yet, and what should be rendered when errors occur. To solve these edge cases, you can use conditional rendering in React. For the first edge case, simply check whether an `organization` is present or not.

```
class App extends Component {
  ...

  render() {
    const { path, organization, errors } = this.state;

    return (
      <div>
        ...

        <hr />

        {organization ? (
          <Organization organization={organization} errors={errors} />
        ) : (
          <p>No information yet ...</p>
        )}
      </div>
    );
  }
}
```

For the second edge case, you have passed the errors to the Organization component. In case there are errors, it should simply render the error message of each error. Otherwise, it should render the organization. There can be multiple errors regarding different fields and circumstances in GraphQL.

```
const Organization = ({ organization, errors }) => {
  if (errors) {
    return (
      <p>
        <strong>Something went wrong:</strong>
        {errors.map(error => error.message).join(' ')}
      </p>
    );
  }

  return (
    <div>
      <p>
        <strong>Issues from Organization:</strong>
```

```
        <a href={organization.url}>{organization.name}</a>
      </p>
    </div>
  );
};
```

You performed your first GraphQL query in a React application, a plain HTTP POST request with a query as payload. You used a configured axios client instance for it. Afterward, you were able to store the result in React's local state to display it later.

GraphQL Nested Objects in React

Next, we'll request a nested object for the organization. Since the application will eventually show the issues in a repository, you should fetch a repository of an organization as the next step. Remember, a query reaches into the GraphQL graph, so we can nest the `repository` field in the `organization` when the schema defined the relationship between these two entities.

```
const GET_REPOSITORY_OF_ORGANIZATION = `
  {
    organization(login: "the-road-to-learn-react") {
      name
      url
      repository(name: "the-road-to-learn-react") {
        name
        url
      }
    }
  }
`;

class App extends Component {
  ...

  onFetchFromGitHub = () => {
    axiosGitHubGraphQL
      .post('', { query: GET_REPOSITORY_OF_ORGANIZATION })
      .then(result =>
          ...
      );
  };

  ...
}
```

In this case, the repository name is identical to the organization. That's okay for now. Later on, you can define an organization and repository on your own dynamically. In the second step, you can

extend the Organization component with another Repository component as child component. The result for the query should now have a nested repository object in the organization object.

```
const Organization = ({ organization, errors }) => {
  if (errors) {
    ...
  }

  return (
    <div>
      <p>
        <strong>Issues from Organization:</strong>
        <a href={organization.url}>{organization.name}</a>
      </p>
      <Repository repository={organization.repository} />
    </div>
  );
};

const Repository = ({ repository }) => (
  <div>
    <p>
      <strong>In Repository:</strong>
      <a href={repository.url}>{repository.name}</a>
    </p>
  </div>
);
```

The GraphQL query structure aligns perfectly with your component tree. It forms a natural fit to continue extending the query structure like this, by nesting other objects into the query, and extending the component tree along the structure of the GraphQL query. Since the application is an issue tracker, we need to add a list field of issues to the query.

If you want to follow the query structure more thoughtfully, open the "Docs" sidebar in GraphiQL to learn out about the types `Organization`, `Repository`, `Issue`. The paginated issues list field can be found there as well. It's always good to have an overview of the graph structure.

Now let's extend the query with the list field for the issues. These issues are a paginated list in the end. We will cover these more later; for now, nest it in the `repository` field with a `last` argument to fetch the last items of the list.

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

```
        name
        url
        issues(last: 5) {
          edges {
            node {
              id
              title
              url
            }
          }
        }
      }
    }
  }
`;
```

You can also request an id for each issue using the `id` field on the issue's `node` field, to use a `key` attribute for your list of rendered items in the component, which is considered best practice in React. Remember to adjust the name of the query variable when its used to perform the request.

```
class App extends Component {
  ...

  onFetchFromGitHub = () => {
    axiosGitHubGraphQL
      .post('', { query: GET_ISSUES_OF_REPOSITORY })
      .then(result =>
          ...
      );
  };

  ...
}
```

The component structure follows the query structure quite naturally again. You can add a list of rendered issues to the Repository component. It is up to you to extract it to its own component as a refactoring to keep your components concise, readable, and maintainable.

```
const Repository = ({ repository }) => (
  <div>
    <p>
      <strong>In Repository:</strong>
      <a href={repository.url}>{repository.name}</a>
    </p>

    <ul>
      {repository.issues.edges.map(issue => (
        <li key={issue.node.id}>
          <a href={issue.node.url}>{issue.node.title}</a>
```

```
          </li>
        ))}
      </ul>
    </div>
  );
```

That's it for the nested objects, fields, and list fields in a query. Once you run your application again, you should see the last issues of the specified repository rendered in your browser.

GraphQL Variables and Arguments in React

Next we'll make use of the form and input elements. They should be used to request the data from GitHub's GraphQL API when a user fills in content and submits it. The content is also used for the initial request in `componentDidMount()` of the App component. So far, the organization `login` and repository `name` were inlined arguments in the query. Now, you should be able to pass in the `path` from the local state to the query to define dynamically an organization and repository. That's where variables in a GraphQL query came into play, do you remember?

First, let's use a naive approach by performing string interpolation with JavaScript rather than using GraphQL variables. To do this, refactor the query from a template literal variable to a function that returns a template literal variable. By using the function, you should be able to pass in an organization and repository.

```
const getIssuesOfRepositoryQuery = (organization, repository) => `
  {
    organization(login: "${organization}") {
      name
      url
      repository(name: "${repository}") {
        name
        url
        issues(last: 5) {
          edges {
            node {
              id
              title
              url
            }
          }
        }
      }
    }
  }
`;
```

Next, call the `onFetchFromGitHub()` class method in the submit handle, but also when the component mounts in `componentDidMount()` with the initial local state of the `path` property. These are the two essential places to fetch the data from the GraphQL API on initial render, and on every other manual submission from a button click.

```
class App extends Component {
  state = {
    path: 'the-road-to-learn-react/the-road-to-learn-react',
    organization: null,
    errors: null,
  };

  componentDidMount() {
    this.onFetchFromGitHub(this.state.path);
  }

  onChange = event => {
    this.setState({ path: event.target.value });
  };

  onSubmit = event => {
    this.onFetchFromGitHub(this.state.path);

    event.preventDefault();
  };

  onFetchFromGitHub = () => {
    ...
  }

  render() {
    ...
  }
}
```

Lastly, call the function that returns the query instead of passing the query string directly as payload. Use the JavaScript's split method on a string to retrieve the prefix and suffix of the `/` character from the path variable where the prefix is the organization and the suffix is the repository.

```
class App extends Component {
  ...

  onFetchFromGitHub = path => {
    const [organization, repository] = path.split('/');

    axiosGitHubGraphQL
      .post('', {
        query: getIssuesOfRepositoryQuery(organization, repository),
```

```
      })
      .then(result =>
        this.setState(() => ({
          organization: result.data.data.organization,
          errors: result.data.errors,
        })),
      );
  };

  ...
}
```

Since the split returns an array of values and it is assumed that there is only one slash in the path, the array should consist of two values: the organization and the repository. That's why it is convenient to use a JavaScript array destructuring to pull out both values from an array in the same line.

Note that the application is not built for to be robust, but is intended only as a learning experience. It is unlikely anyone will ask a user to input the organization and repository with a different pattern than *organization/repository*, so there is no validation included yet. Still, it is a good foundation for you to gain experience with the concepts.

If you want to go further, you can extract the first part of the class method to its own function, which uses axios to send a request with the query and return a promise. The promise can be used to resolve the result into the local state, using `this.setState()` in the `then()` resolver block of the promise.

```
const getIssuesOfRepository = path => {
  const [organization, repository] = path.split('/');

  return axiosGitHubGraphQL.post('', {
    query: getIssuesOfRepositoryQuery(organization, repository),
  });
};

class App extends Component {
  ...

  onFetchFromGitHub = path => {
    getIssuesOfRepository(path).then(result =>
      this.setState(() => ({
        organization: result.data.data.organization,
        errors: result.data.errors,
      })),
    );
  };

  ...
}
```

You can always split your applications into parts, be they functions or components, to make them concise, readable, reusable and testable. The function that is passed to `this.setState()` can be extracted as higher-order function. It has to be a higher-order function, because you need to pass the result of the promise, but also provide a function for the `this.setState()` method.

```
const resolveIssuesQuery = queryResult => () => ({
  organization: queryResult.data.data.organization,
  errors: queryResult.data.errors,
});

class App extends Component {
  ...

  onFetchFromGitHub = path => {
    getIssuesOfRepository(path).then(queryResult =>
      this.setState(resolveIssuesQuery(queryResult)),
    );
  };

  ...
}
```

Now you've made your query flexible by providing dynamic arguments to your query. Try it by starting your application on the command line and by filling in a different organization with a specific repository (e.g. *facebook/create-react-app*).

It's a decent setup, but there was nothing to see about variables yet. You simply passed the arguments to the query using a function and string interpolation with template literals. Now we'll use GraphQL variables instead, to refactor the query variable again to a template literal that defines inline variables.

```
const GET_ISSUES_OF_REPOSITORY = `
  query ($organization: String!, $repository: String!) {
    organization(login: $organization) {
      name
      url
      repository(name: $repository) {
        name
        url
        issues(last: 5) {
          edges {
            node {
              id
              title
              url
            }
```

```
            }
          }
        }
      }
    }
  `;
```

Now you can pass those variables as arguments next to the query for the HTTP POST request:

```
const getIssuesOfRepository = path => {
  const [organization, repository] = path.split('/');

  return axiosGitHubGraphQL.post('', {
    query: GET_ISSUES_OF_REPOSITORY,
    variables: { organization, repository },
  });
};
```

Finally, the query takes variables into account without detouring into a function with string interpolation. I strongly suggest practicing with the exercises below before continuing to the next section. We've yet to discuss features like fragments or operation names, but we'll soon cover them using Apollo instead of plain HTTP with axios.

Exercises:

- Confirm your source code for the last section
  - Confirm the changes from the last section
- Explore and add fields to your organization, repository and issues
  - Extend your components to display the additional information
- Read more about serving a GraphQL API over HTTP

REACT GRAPHQL PAGINATION

Last section you implemented a list field in your GraphQL query, which fit into the flow of structuring the query with nested objects and a list responsible for showing partial results of the query in React.

In this section, you will explore pagination with list fields with GraphQL in React in more detail. Initially, you will learn more about the arguments of list fields. Further, you will add one more nested list field to your query. Finally, you will fetch another page of the paginated `issues` list with your query.

Let's start by extending the `issues` list field in your query with one more argument:

```
const GET_ISSUES_OF_REPOSITORY = `
  query ($organization: String!, $repository: String!) {
    organization(login: $organization) {
      name
      url
      repository(name: $repository) {
        name
        url
        issues(last: 5, states: [OPEN]) {
          edges {
            node {
              id
              title
              url
            }
          }
        }
      }
    }
  }
`;
```

If you read the arguments for the `issues` list field using the "Docs" sidebar in GraphiQL, you can explore which arguments you can pass to the field. One of these is the `states` argument, which defines whether or not to fetch open or closed issues. The previous implementation of the query has shown you how to refine the list field, in case you only want to show open issues. You can explore more arguments for the `issues` list field, but also for other list fields, using the documentation from Github's API.

Now we'll implement another nested list field that could be used for pagination. Each issue in a repository can have reactions, essentially emoticons like a smiley or a thumbs up. Reactions can be seen as another list of paginated items. First, extend the query with the nested list field for reactions:

```
const GET_ISSUES_OF_REPOSITORY = `
  query ($organization: String!, $repository: String!) {
    organization(login: $organization) {
      name
      url
      repository(name: $repository) {
        name
        url
        issues(last: 5, states: [OPEN]) {
          edges {
            node {
              id
```

```
              title
              url
              reactions(last: 3) {
                edges {
                  node {
                    id
                    content
                  }
                }
              }
            }
          }
        }
      }
    }
  }
`;
```

Second, render the list of reactions in one of your React components again. Implement dedicated List and Item components, such as ReactionsList and ReactionItem for it. As an exercise, try keeping the code for this application readable and maintainable.



```
const Repository = ({ repository }) => (
  <div>
    ...

    <ul>
      {repository.issues.edges.map(issue => (
        <li key={issue.node.id}>
          <a href={issue.node.url}>{issue.node.title}</a>

          <ul>
            {issue.node.reactions.edges.map(reaction => (
              <li key={reaction.node.id}>{reaction.node.content}</li>
            ))}
          </ul>
        </li>
      ))}
    </ul>
  </div>
);
```

You extended the query and React's component structure to render the result. It's a straightforward implementation when you are using a GraphQL API as your data source which has a well defined underlying schema for these field relationships.

Lastly, you will implement real pagination with the `issues` list field, as there should be a button to fetch more issues from the GraphQL API to make it a function of a completed application. Here is how to implement a button:

```
const Repository = ({
  repository,
  onFetchMoreIssues,
}) => (
  <div>
    ...

    <ul>
      ...
    </ul>

    <hr />

    <button onClick={onFetchMoreIssues}>More</button>
  </div>
);
```

The handler for the button passes through all the components to reach the Repository component:

```
const Organization = ({
  organization,
  errors,
  onFetchMoreIssues,
}) => {
  ...

  return (
    <div>
      <p>
        <strong>Issues from Organization:</strong>
        <a href={organization.url}>{organization.name}</a>
      </p>
      <Repository
        repository={organization.repository}
        onFetchMoreIssues={onFetchMoreIssues}
      />
    </div>
  );
};
```

Logic for the function is implemented in the App component as class method. It passes to the Organization component as well.

```
class App extends Component {
  ...

  onFetchMoreIssues = () => {
```

```
    ...
  };

  render() {
    const { path, organization, errors } = this.state;

    return (
      <div>
        ...

        {organization ? (
          <Organization
            organization={organization}
            errors={errors}
            onFetchMoreIssues={this.onFetchMoreIssues}
          />
        ) : (
          <p>No information yet ...</p>
        )}
      </div>
    );
  }
}
```

Before implementing the logic for it, there needs to be a way to identify the next page of the paginated list. To extend the inner fields of a list field with fields for meta information such as the `pageInfo` or the `totalCount` information, use `pageInfo` to define the next page on button-click. Also, the `totalCount` is only a nice way to see how many items are in the next list:

```
const GET_ISSUES_OF_REPOSITORY = `
  query ($organization: String!, $repository: String!) {
    organization(login: $organization) {
      name
      url
      repository(name: $repository) {
        ...
        issues(last: 5, states: [OPEN]) {
          edges {
            ...
          }
          totalCount
          pageInfo {
            endCursor
            hasNextPage
          }
        }
      }
    }
  }
`;
```

Now, you can use this information to fetch the next page of issues by providing the cursor as a variable to your query. The cursor, or the `after` argument, defines the starting point to fetch more items from the paginated list.

```
class App extends Component {
  ...

  onFetchMoreIssues = () => {
    const {
      endCursor,
    } = this.state.organization.repository.issues.pageInfo;

    this.onFetchFromGitHub(this.state.path, endCursor);
  };

  ...
}
```

The second argument wasn't introduced to the `onFetchFromGitHub()` class method yet. Let's see how that turns out.

```
const getIssuesOfRepository = (path, cursor) => {
  const [organization, repository] = path.split('/');

  return axiosGitHubGraphQL.post('', {
    query: GET_ISSUES_OF_REPOSITORY,
    variables: { organization, repository, cursor },
  });
};

class App extends Component {
  ...

  onFetchFromGitHub = (path, cursor) => {
    getIssuesOfRepository(path, cursor).then(queryResult =>
      this.setState(resolveIssuesQuery(queryResult, cursor)),
    );
  };

  ...
}
```

The argument is simply passed to the `getIssuesOfRepository()` function, which makes the GraphQL API request, and returns the promise with the query result. Check the other functions that call the `onFetchFromGitHub()` class method, and notice how they don't make use of the second argument, so the cursor parameter will be `undefined` when it's passed to the GraphQL

API call. Either the query uses the cursor as argument to fetch the next page of a list, or it fetches the initial page of a list by having the cursor not defined at all:

```
const GET_ISSUES_OF_REPOSITORY = `
  query (
    $organization: String!,
    $repository: String!,
    $cursor: String
  ) {
    organization(login: $organization) {
      name
      url
      repository(name: $repository) {
        ...
        issues(first: 5, after: $cursor, states: [OPEN]) {
          edges {
            ...
          }
          totalCount
          pageInfo {
            endCursor
            hasNextPage
          }
        }
      }
    }
  }
`;
```

In the previous template string, the `cursor` is passed as variable to the query and used as `after` argument for the list field. The variable is not enforced though, because there is no exclamation mark next to it, so it can be `undefined`. This happens for the initial page request for a paginated list, when you only want to fetch the first page. Further, the argument `last` has been changed to `first` for the `issues` list field, because there won't be another page after you fetched the last item in the initial request. Thus, you have to start with the first items of the list to fetch more items until you reach the end of the list.

That's it for fetching the next page of a paginated list with GraphQL in React, except one final step. Nothing updates the local state of the App component about a page of issues yet, so there are still only the issues from the initial request. You want to merge the old pages of issues with the new page of issues in the local state of the App component, while keeping the organization and repository information in the deeply nested state object intact. The perfect time for doing this is when the promise for the query resolves. You already extracted it as a function outside of the App component, so you can use this place to handle the incoming result and return a result with your own structure and information. Keep in mind that the incoming result can be an initial request

when the App component mounts for the first time, or after a request to fetch more issues happens, such as when the "More" button is clicked.

```
const resolveIssuesQuery = (queryResult, cursor) => state => {
  const { data, errors } = queryResult.data;

  if (!cursor) {
    return {
      organization: data.organization,
      errors,
    };
  }

  const { edges: oldIssues } = state.organization.repository.issues;
  const { edges: newIssues } = data.organization.repository.issues;
  const updatedIssues = [...oldIssues, ...newIssues];

  return {
    organization: {
      ...data.organization,
      repository: {
        ...data.organization.repository,
        issues: {
          ...data.organization.repository.issues,
          edges: updatedIssues,
        },
      },
    },
    errors,
  };
};
```

The function is a complete rewrite, because the update mechanism is more complex now. First, you passed the `cursor` as an argument to the function, which determines whether it was an initial query or a query to fetch another page of issues. Second, if the `cursor` is `undefined`, the function can return early with the state object that encapsulates the plain query result, same as before. There is nothing to keep intact in the state object from before, because it is an initial request that happens when the App component mounts or when a user submits another request which should overwrite the old state anyway. Third, if it is a fetch more query and the cursor is there, the old and new issues from the state and the query result get merged in an updated list of issues. In this case, a JavaScript destructuring alias is used to make naming both issue lists more obvious. Finally, the function returns the updated state object. Since it is a deeply nested object with multiple levels to update, use the JavaScript spread operator syntax to update each level with a new query result. Only the `edges` property should be updated with the merged list of issues.

Next, use the `hasNextPage` property from the `pageInfo` that you requested to show a "More" button (or not). If there are no more issues in the list, the button should disappear.

```
const Repository = ({ repository, onFetchMoreIssues }) => (
  <div>
    ...

    <hr />

    {repository.issues.pageInfo.hasNextPage && (
      <button onClick={onFetchMoreIssues}>More</button>
    )}
  </div>
);
```

Now you've implemented pagination with GraphQL in React. For practice, try more arguments for your issues and reactions list fields on your own. Check the "Docs" sidebar in GraphiQL to find out about arguments you can pass to list fields. Some arguments are generic, but have arguments that are specific to lists. These arguments should show you how finely-tuned requests can be with a GraphQL query.

Exercises:

- Confirm your source code for the last section
- Confirm the changes from the last section
- Explore further arguments, generic or specific for the type, on the `issues` and `reactions` list fields
- Think about ways to beautify the updating mechanism of deeply nested state objects and contribute your thoughts to it

---

REACT GRAPHQL MUTATION

You fetched a lot of data using GraphQL in React, the larger part of using GraphQL. However, there are always two sides to such an interface: read and write. That's where GraphQL mutations complement the interface. Previously, you learned about GraphQL mutations using GraphiQL without React. In this section, you will implement such a mutation in your React GraphQL application.

You have executed GitHub's `addStar` mutation before in GraphiQL. Now, let's implement this mutation in React. Before implementing the mutation, you should query additional information about the repository, which is partially required to star the repository in a mutation.

```
const GET_ISSUES_OF_REPOSITORY = `
  query (
```

```
    $organization: String!,
    $repository: String!,
    $cursor: String
  ) {
    organization(login: $organization) {
      name
      url
      repository(name: $repository) {
        id
        name
        url
        viewerHasStarred
        issues(first: 5, after: $cursor, states: [OPEN]) {
          ...
        }
      }
    }
  }
`;
```

The `viewerHasStarred` field returns a boolean that tells whether the viewer has starred the repository or not. This boolean helps determine whether to execute a `addStar` or `removeStar` mutation in the next steps. For now, you will only implement the `addStar` mutation. The `removeStar` mutation will be left off as part of the exercise. Also, the `id` field in the query returns the identifier for the repository, which you will need to clarify the target repository of your mutation.

The best place to trigger the mutation is a button that stars or unstars the repository. That's where the `viewerHasStarred` boolean can be used for a conditional rendering to show either a "Star" or "Unstar" button. Since you are going to star a repository, the Repository component is the best place to trigger the mutation.

```
const Repository = ({
  repository,
  onFetchMoreIssues,
  onStarRepository,
}) => (
  <div>
    ...

    <button
      type="button"
      onClick={() => onStarRepository()}
    >
      {repository.viewerHasStarred ? 'Unstar' : 'Star'}
    </button>

    <ul>
      ...
```

```
      </ul>
    </div>
  );
```

To identify the repository to be starred, the mutation needs to know about the `id` of the repository. Pass the `viewerHasStarred` property as a parameter to the handler, since you'll use the parameter to determine whether you want to execute the star or unstar mutation later.

```
const Repository = ({ repository, onStarRepository }) => (
  <div>
    ...

    <button
      type="button"
      onClick={() =>
        onStarRepository(repository.id, repository.viewerHasStarred)
      }
    >
      {repository.viewerHasStarred ? 'Unstar' : 'Star'}
    </button>

    ...
  </div>
);
```

The handler should be defined in the App component. It passes through each component until it reaches the Repository component, also reaching through the Organization component on its way.

```
const Organization = ({
  organization,
  errors,
  onFetchMoreIssues,
  onStarRepository,
}) => {
  ...

  return (
    <div>
      ...
      <Repository
        repository={organization.repository}
        onFetchMoreIssues={onFetchMoreIssues}
        onStarRepository={onStarRepository}
      />
    </div>
  );
};
```

Now it can be defined in the App component. Note that the `id` and the `viewerHasStarred` information can be destructured from the App's local state, too. This is why you wouldn't need to pass this information in the handler, but use it from the local state instead. However, since the Repository component knew about the information already, it is fine to pass the information in the handler, which also makes the handler more explicit. It's also good preparation for dealing with multiple repositories and repository components later, since the handler will need to be more specific in these cases.

```
class App extends Component {
  ...

  onStarRepository = (repositoryId, viewerHasStarred) => {
    ...
  };

  render() {
    const { path, organization, errors } = this.state;

    return (
      <div>
        ...

        {organization ? (
          <Organization
            organization={organization}
            errors={errors}
            onFetchMoreIssues={this.onFetchMoreIssues}
            onStarRepository={this.onStarRepository}
          />
        ) : (
          <p>No information yet ...</p>
        )}
      </div>
    );
  }
}
```

Now, you can implement the handler. The mutation can be outsourced from the component. Later, you can use the `viewerHasStarred` boolean in the handler to perform a `addStar` or `removeStar` mutation. Executing the mutation looks similar to the GraphQL query from before. The API endpoint is not needed, because it was set in the beginning when you configured axios. The mutation can be sent in the `query` payload, which we'll cover later. The `variables` property is optional, but you need to pass the identifier.

```
const addStarToRepository = repositoryId => {
  return axiosGitHubGraphQL.post('', {
```

```
    query: ADD_STAR,
    variables: { repositoryId },
  });
};

class App extends Component {
  ...

  onStarRepository = (repositoryId, viewerHasStarred) => {
    addStarToRepository(repositoryId);
  };

  ...
}
```

Before you define the `addStar` mutation, check GitHub's GraphQL API again. There, you will find all information about the structure of the mutation, the required arguments, and the available fields for the result. For instance, you can include the `viewerHasStarred` field in the returned result to get an updated boolean of a starred or unstarred repository.

```
const ADD_STAR = `
  mutation ($repositoryId: ID!) {
    addStar(input:{starrableId:$repositoryId}) {
      starrable {
        viewerHasStarred
      }
    }
  }
`;
```

You could already execute the mutation in the browser by clicking the button. If you haven't starred the repository before, it should be starred after clicking the button. You can visit the repository on GitHub to get visual feedback, though you won't see any results reflected yet. The button still shows the "Star" label when the repository wasn't starred before, because the `viewerHasStarred` boolean wasn't updated in the local state of the App component after the mutation. That's the next thing you are going to implement. Since axios returns a promise, you can use the `then()` method on the promise to resolve it with your own implementation details.

```
const resolveAddStarMutation = mutationResult => state => {
  ...
};

class App extends Component {
  ...

  onStarRepository = (repositoryId, viewerHasStarred) => {
    addStarToRepository(repositoryId).then(mutationResult =>
```

```
        this.setState(resolveAddStarMutation(mutationResult)),
      );
    };

    ...
  }
```

When resolving the promise from the mutation, you can find out about the `viewerHasStarred` property in the result. That's because you defined this property as a field in your mutation. It returns a new state object for React's local state, because you used the function in `this.setState()`. The spread operator syntax is used here, to update the deeply nested data structure. Only the `viewerHasStarred` property changes in the state object, because it's the only property returned by the resolved promise from the successful request. All other parts of the local state stay intact.

```
const resolveAddStarMutation = mutationResult => state => {
  const {
    viewerHasStarred,
  } = mutationResult.data.data.addStar.starrable;

  return {
    ...state,
    organization: {
      ...state.organization,
      repository: {
        ...state.organization.repository,
        viewerHasStarred,
      },
    },
  };
};
```

Now try to star the repository again. You may have to go on the GitHub page and unstar it first. The button label should adapt to the updated `viewerHasStarred` property from the local state to show a "Star" or "Unstar" label. You can use what you've learned about starring repositories to implement a `removeStar` mutation.

We also want to show the current number of people who have starred the repository, and update this count in the `addStar` and `removeStar` mutations. First, retrieve the total count of stargazers by adding the following fields to your query:

```
const GET_ISSUES_OF_REPOSITORY = `
  query (
    $organization: String!,
    $repository: String!,
```

```
    $cursor: String
  ) {
    organization(login: $organization) {
      name
      url
      repository(name: $repository) {
        id
        name
        url
        stargazers {
          totalCount
        }
        viewerHasStarred
        issues(first: 5, after: $cursor, states: [OPEN]) {
          ...
        }
      }
    }
  }
`;
```

Second, you can show the count as a part of your button label:

```
const Repository = ({
  repository,
  onFetchMoreIssues,
  onStarRepository,
}) => (
  <div>
    ...

    <button
      type="button"
      onClick={() =>
        onStarRepository(repository.id, repository.viewerHasStarred)
      }
    >
      {repository.stargazers.totalCount}
      {repository.viewerHasStarred ? ' Unstar' : ' Star'}
    </button>

    <ul>
      ...
    </ul>
  </div>
);
```

Now we want the count to update when you star (or unstar) a repository. It is the same issue as the missing update for the `viewerHasStarred` property in the local state of the component after the `addStar` mutation succeeded. Return to your mutation resolver and update the total count of

stargazers there as well. While the stargazer object isn't returned as a result from the mutation, you can increment and decrement the total count after a successful mutation manually using a counter along with the `addStar` mutation.

```
const resolveAddStarMutation = mutationResult => state => {
  const {
    viewerHasStarred,
  } = mutationResult.data.data.addStar.starrable;

  const { totalCount } = state.organization.repository.stargazers;

  return {
    ...state,
    organization: {
      ...state.organization,
      repository: {
        ...state.organization.repository,
        viewerHasStarred,
        stargazers: {
          totalCount: totalCount + 1,
        },
      },
    },
  };
};
```



You have implemented your first mutation in React with GraphQL. So far, you have just implemented the `addStar` mutation. Even though the button already reflects the `viewerHasStarred` boolean by showing a "Star" or "Unstar" label, the button showing "Unstar" should still execute the `addStar` mutation. The `removeStar` mutation to unstar the repository is one of the practice exercises mentioned below.

Exercises:

- Confirm your source code for the last section
  - Confirm the changes from the last section
- Implement the `removeStar` mutation, which is used analog to the `addStar` mutation.
  - The `onStarRepository` class method has already access to the `viewerHasStarred` property.
  - Conditionally execute a `addStar` or `removeStar` mutation in the class handler.
  - Resolve the new state after removing a star from a repository.
  - Align your final thoughts with this implementation.
- Implement the `addReaction` mutation for an issue
- Implement more fine-grained components (e.g. IssueList, IssueItem, ReactionList, ReactionItem)

- Extract components to their own files and use import and export statements to use them again in the App or other extracted components

---

SHORTCOMINGS OF GRAPHQL IN REACT WITHOUT A GRAPHQL CLIENT LIBRARY

We implemented a simple GitHub issue tracker that uses React and GraphQL without a dedicated library for GraphQL, using only axios to communicate with the GraphQL API with HTTP POST methods. I think it is important to work with raw technologies, in this case GraphQL, using plain HTTP methods, before introducing another abstraction. The Apollo library offers an abstraction that makes using GraphQL in React much easier, so you will use Apollo for your next application. For now, using GraphQL with HTTP has shown you two important things before introducing Apollo:

- How GraphQL works when using a puristic interface such as HTTP.
- The shortcomings of using no sophisticated GraphQL Client library in React, because you have to do everything yourself.

Before we move on, I want to address the shortcomings of using puristic HTTP methods to read and write data to your GraphQL API in a React application:

- **Complementary:** To call a GraphQL API from your client application, use HTTP methods. There are several quality libraries out there for HTTP requests, one of which is axios. That's why you have used axios for the previous application. However, using axios (or any other HTTP client library) doesn't feel like the best fit to complement a GraphQL centred interface. For instance, GraphQL doesn't use the full potential of HTTP. It's just fine to default to HTTP POST and only one API endpoint. It doesn't use resources and methods on those resources like a RESTful interface, so it makes no sense to specify a HTTP method and an API endpoint with every request, but to set it up once in the beginning instead. GraphQL comes with its own constraints. You could see it as a layer on top of HTTP when it's not as important for a developer to know about the underlying HTTP.

- **Declarative:** Every time you make a query or mutation when using plain HTTP requests, you have to make a dedicated call to the API endpoint using a library such as axios. It's an imperative way of reading and writing data to your backend. However, what if there was a declarative approach to making queries and mutations? What if there was a way to co-locate queries and mutations to your view-layer components? In the previous application, you experienced how the query shape aligned perfectly with your component hierarchy shape. What if the queries and mutations would align in the same way? That's the power of co-locating your data-layer with

your view-layer, and you will find out more about it when you use a dedicated GraphQL client library for it.

- **Feature Support:** When using plain HTTP requests to interact with your GraphQL API, you are not leveraging the full potential of GraphQL. Imagine you want to split your query from the previous application into multiple queries that are co-located with their respective components where the data is used. That's when GraphQL would be used in a declarative way in your view-layer. But when you have no library support, you have to deal with multiple queries on your own, keeping track of all of them, and trying to merge the results in your state-layer. If you consider the previous application, splitting up the query into multiple queries would add a whole layer of complexity to the application. A GraphQL client library deals with aggregating the queries for you.

- **Data Handling:** The naive way for data handling with puristic HTTP requests is a subcategory of the missing feature support for GraphQL when not using a dedicated library for it. There is no one helping you out with normalizing your data and caching it for identical requests. Updating your state-layer when resolving fetched data from the data-layer becomes a nightmare when not normalizing the data in the first place. You have to deal with deeply nested state objects which lead to the verbose usage of the JavaScript spread operator. When you check the implementation of the application in the GitHub repository again, you will see that the updates of React's local state after a mutation and query are not nice to look at. A normalizing library such as normalizr could help you to improve the structure of your local state. You learn more about normalizing your state in the book The Road to Redux. In addition to a lack of caching and normalizing support, avoiding libraries means missing out on functionalities for pagination and optimistic updates. A dedicated GraphQL library makes all these features available to you.

- **GraphQL Subscriptions:** While there is the concept of a query and mutation to read and write data with GraphQL, there is a third concept of a GraphQL **subscription** for receiving real-time data in a client-sided application. When you would have to rely on plain HTTP requests as before, you would have to introduce WebSockets next to it. It enables you to introduce a long-lived connection for receiving results over time. In conclusion, introducing GraphQL subscriptions would add another tool to your application. However, if you would introduce a GraphQL library for it on the client-side, the library would probably implement GraphQL subscriptions for you.

  I am looking forward to introducing Apollo as a GraphQL client library to your React application. It will help with the aforementioned shortcomings. However, I do strongly believe it was good to learn about GraphQL in React without a GraphQL library in the beginning.

. . .

You can find the final repository on GitHub. The repository showcases most of the exercise tasks too. The application is not feature complete since it doesn't cover all edge cases and isn't styled.

However, I hope the implementation walkthrough with plain GraphQL in React has helped you to understand using only GraphQL client-side in React using HTTP requests. I feel it's important to take this step before using a sophisticated GraphQL client library such as Apollo or Relay.

I've shown how to implement a React application with GraphQL and HTTP requests without using a library like Apollo. Next, you will continue learning about using GraphQL in React using Apollo instead of basic HTTP requests with axios. The Apollo GraphQL Client makes caching your data, normalizing it, performing optimistic updates, and pagination effortless. That's not all by a long shot, so stay tuned for the next applications you are are going to build with GraphQL.

This tutorial is part 3 of 5 in this series.

Show Comments

Part 1: Getting Started with GitHub's GraphQL API
Part 2: GraphQL Tutorial for Beginners
Part 4: Apollo Client Tutorial for Beginners
Part 5: React with Apollo and GraphQL Tutorial

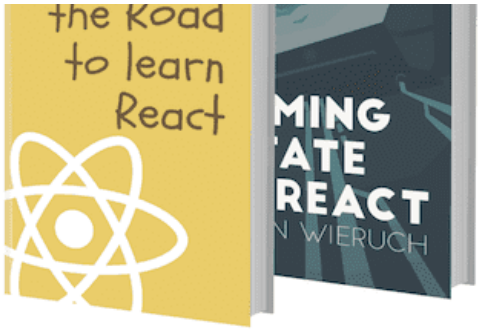
KEEP READING ABOUT REACT ›

WRITING TESTS FOR APOLLO CLIENT IN REACT

In a previous application, you have learned how to mock a GraphQL server in different ways when having Apollo Client as GraphQL client in your React application. The following application shows you...

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

Contact Me Privacy & Terms