GraphQL Clients

Since a GraphQL API has more underlying structure than a REST API, there are more powerful clients like Relay which can automatically handle batching, caching, and other features. But you don't need a complex client to call a GraphQL server. With `express-graphql`, you can just send an HTTP POST request to the endpoint you mounted your GraphQL server on, passing the GraphQL query as the `query` field in a JSON payload.

For example, let's say we mounted a GraphQL server on http://localhost:4000/graphql as in the example code for running an Express GraphQL server, and we want to send the GraphQL query `{ hello }`. We can do this from the command line with `curl`. If you paste this into a terminal:

```
curl -X POST \
-H "Content-Type: application/json" \
-d '{"query": "{ hello }"}' \
http://localhost:4000/graphql
```

You should see the output returned as JSON:

```
{"data":{"hello":"Hello world!"}}
```

If you prefer to use a graphical user interface to send a test query, you can use clients such as GraphiQL and Insomnia.

It's also simple to send GraphQL from the browser. Open up http://localhost:4000/graphql, open a developer console, and paste in:

```
fetch('/graphql', {
  method: 'POST',
  headers: {
    'Content-Type': 'application/json',
    'Accept': 'application/json',
  },
  body: JSON.stringify({query: "{ hello }"})
})
  .then(r => r.json())
  .then(data => console.log('data returned:', data));
```

You should see the data returned, logged in the console:

```
data returned: Object { hello: "Hello world!" }
```

In this example, the query was just a hardcoded string. As your application becomes more complex, and you add GraphQL endpoints that take arguments as described in Passing Arguments, you will want to construct GraphQL queries using variables in client code. You can do this by including a keyword prefixed with a dollar sign in the query, and passing an extra `variables` field on the payload.

For example, let's say you're running the example server from Passing Arguments that has a schema of

```
type Query {
  rollDice(numDice: Int!, numSides: Int): [Int]
}
```

You could access this from JavaScript with the code:

```
var dice = 3;
var sides = 6;
var query = `query RollDice($dice: Int!, $sides: Int) {
  rollDice(numDice: $dice, numSides: $sides)
}`;

fetch('/graphql', {
  method: 'POST',
  headers: {
    'Content-Type': 'application/json',
    'Accept': 'application/json',
  },
  body: JSON.stringify({
    query,
    variables: { dice, sides },
  })
})
  .then(r => r.json())
  .then(data => console.log('data returned:', data));
```

Using this syntax for variables is a good idea because it automatically prevents bugs due to escaping, and it makes it easier to monitor your server.

In general, it will take a bit more time to set up a GraphQL client like Relay, but it's worth it to get more features as your application grows. You might want to start out just using HTTP requests as the underlying transport layer, and switching to a more complex client as your application gets more complex.

At this point you can write a client and server in GraphQL for an API that receives a single string. To do more, you will want to learn how to use the other basic data types.

Continue Reading →
Basic Types