**9 Consuming GraphQL APIs**

This chapter covers

* Running a GraphQL mock server to test our API design
* Using the GraphiQL client to explore and consume a GraphQL API
* Running queries and mutations against a GraphQL API
* Consuming a GraphQL API programmatically using cURL and Python

This chapter teaches you how to consume GraphQL APIs. As we learned in chapter 8, GraphQL offers a query language for web APIs, and in this chapter you’ll learn how to use this language to run queries on the server. In particular, you’ll learn how to make queries against a GraphQL API. You’ll learn to explore a GraphQL API to discover its available types, queries, and mutations. Understanding how GraphQL APIs work from the client side is an important step toward mastering GraphQL.

Learning to interact with GraphQL APIs will help you learn to consume the APIs exposed by other vendors, it’ll let you run tests against your own APIs, and it’ll help you design better APIs. You’ll learn to use the GraphiQL client to explore and visualize a GraphQL API. As you’ll see, GraphiQL offers an interactive query panel that makes it easier to run queries on the server.

To illustrate the concepts and ideas behind GraphQL’s query language, we’ll run practical examples using the products API we designed in chapter 8. Since we haven’t implemented the API specification for the products API, we’ll learn to run a mock server—an important part of the API development process, as it makes testing and validating an API design so much easier. Finally, you’ll also learn to run queries against a GraphQL API programmatically using tools such as cURL and Python.

**9.1 Running a GraphQL mock server**

In this section, we explain how we can run a GraphQL mock server to explore and test our API. A mock server is a fake server that emulates the behavior of the real server, offering the same endpoints and capabilities, but using fake data. For example, a mock server for the products API is a server that mimics the implementation of the products API and offers the same interface that we developed in chapter 8.

**DEFINITION** *Mock servers* are fake servers that mimic the behavior of a real server. They are commonly used for developing API clients while the backend is being implemented. You can launch a mock server using the specification for an API. Mock servers return fake data and typically don’t persist data.

Mock servers are instrumental in the development of web APIs since they allow our API consumers to start working on the client-side code while we work on the backend implementation. In this section, we’ll run a mock server on the products API. The only thing we need to run a mock server is the API specification, which we developed in chapter 8. You’ll find the API specification under ch08/schema.graphql in the GitHub repository for this book.

You can choose from among many different libraries to run a GraphQL mock server. In this chapter, we’ll use GraphQL Faker (<https://github.com/APIs-guru/graphql-faker>), which is one of the most popular GraphQL mocking tools. To install GraphQL Faker, run the following command:

$ npm install graphql-faker

This will create a package-lock.json file under your current directory, as well as a node\_ modules folder. package-lock.json contains information about the dependencies installed together with graphql-faker, while node\_modules is the directory where those dependencies are installed. To run the mock server, execute the following command:

$ ./node\_modules/.bin/graphql-faker schema.graphql

GraphQL Faker normally runs on port 9002, and it exposes three endpoints:

* /editor—An interactive editor where you can develop your GraphQL API.
* /graphql—A GraphiQL interface to your GraphQL API. This is the interface we’ll use to explore the API and run our queries.
* /voyager—An interactive display of your API, which helps you understand the relationships and dependencies between your types (see figure 9.1).

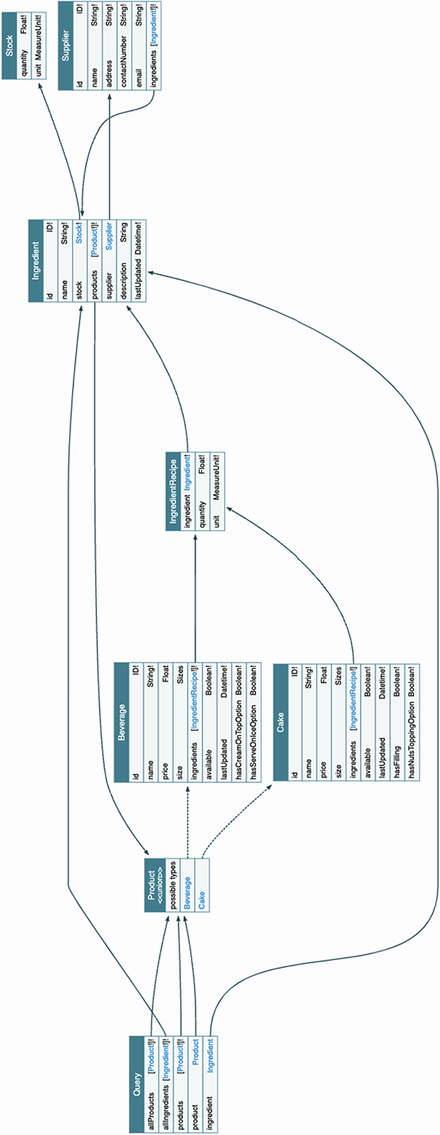


Figure 9.1 Voyager UI for the products API. This UI shows the relationships between object types captured by the queries available in the API. By following the connecting arrows, you can see which objects we can reach from each query.

To start exploring and testing the products API, visit the following address in your browser: http://localhost:9002/graphql (if you’re running GraphQL Faker in a different port, your URL will look different). This endpoint loads a GraphiQL interface for our products API. Figure 9.2 illustrates what this interface looks like and highlights the most important elements in it.

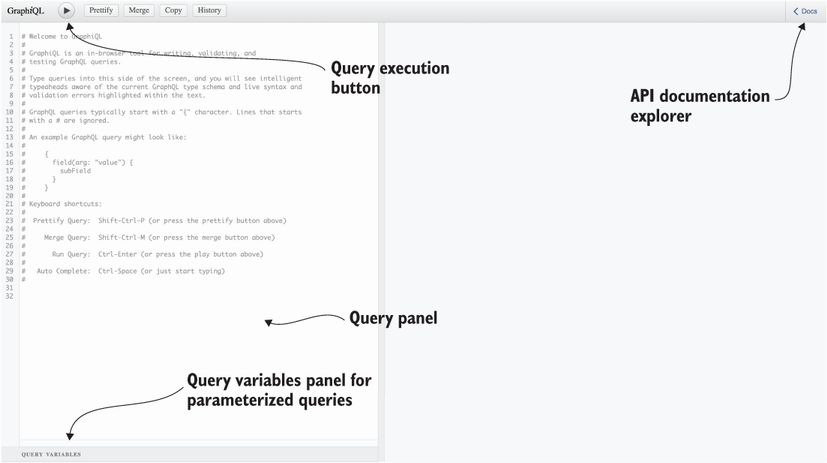


Figure 9.2 API documentation explorer and query panel interface in GraphiQL

To discover the queries and mutations exposed by the API, click the Docs button on the top-right corner of the UI. Upon clicking the Docs button, a side navigation bar will pop up offering two choices: queries or mutations (see figure 9.3 for an illustration). If you select queries, you’ll see the list of queries exposed by the server with their return types. You can click the return types to explore their properties, as you can see in figure 9.3. In the next section, we’ll start testing the GraphQL API!

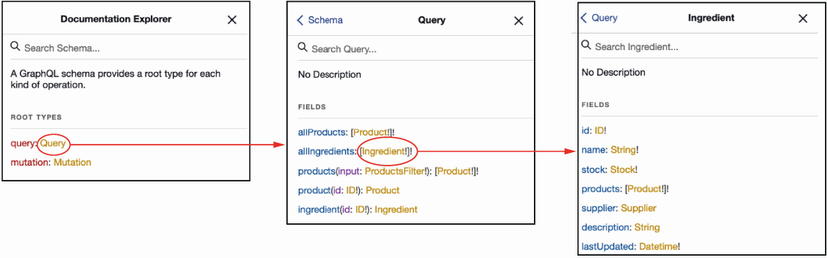


Figure 9.3 By clicking through the Documentation Explorer in GraphiQL, you can inspect all the queries and mutations available in the API, as well as the types they return and their properties.

**9.2 Introducing GraphQL queries**

In this section, we learn to consume a GraphQL API by running queries using GraphiQL. We’ll start with simple queries that don’t require any parameters, and then we’ll move on to queries with parameters.

**9.2.1 Running simple queries**

In this section, we introduce simple queries that don’t take any parameters. The products API offers two queries of this type: allProducts(), which returns a list of all products CoffeeMesh offers, and allIngredients(), which returns a list of all the ingredients.

We’ll use GraphiQL to run queries against the API. To run the query, go to the query editor pane in the GraphiQL UI, which is illustrated in figure 9.2. Listing 9.1 shows how we run the allIngredients() query. As you can see, to run a query we must use the name of the query operation followed by curly braces. Within the curly braces, we declare the selection of properties we want to get from the server. The block within curly braces is called a *selection set*. GraphQL queries must always include a selection set. If you don’t include it, you’ll get an error response from the server. Here, we select only the name of each ingredient. The text representing the query is called a *query document*.

Listing 9.1 Query document running the allIngredients() query

{ ①

allIngredients { ②

name ③

}

}

① We wrap queries within curly braces.

② We run the allIngredients() query.

③ We query the name property.

A response to a successful query from a GraphQL API contains a JSON document with a “data” field, which wraps the query result. An unsuccessful query results in a JSON document that contains an “error” key. Since we’re running a mock server, the API returns random values.

Listing 9.2 Example of successful response for the allIngredients() query

{

"data": { ①

"allIngredients": [ ②

{

"name": "string"

},

{

"name": "string"

}

]

}

}

① A successful response includes a "data" key.

② The result of the query is indexed under a key named after the query itself.

Now that we know the basics of GraphQL queries, let’s spice up our queries by adding parameters!

**9.2.2 Running queries with parameters**

This section explains how we use parameters in GraphQL queries. allIngredients() is a simple query that doesn’t take any parameters. Now let’s see how we can run a query that requires a parameter. One example of such a query is the ingredient() query, which requires an id parameter. The following code shows how we can call the ingredient() query with a random ID. As you can see, we include the query parameters as key-value pairs separated by a colon within parentheses.

Listing 9.3 Running a query with a required parameter

{

ingredient(id: "asdf") { ①

name

}

}

① We call ingredient() with the ID parameter set to "asdf ".

Now that we know how to run queries with parameters, let’s look at the kinds of problems we can run into when running queries and how to deal with them.

**9.2.3 Understanding query errors**

This section explains some of the most common errors you’ll find when running GraphQL queries, and it teaches you how to read and interpret them.

If you omit the required parameter when running the ingredient() query, you’ll get an error from the API. Error responses include an error key pointing to a list of all the errors found by the server. Each error is an object with the following keys:

* message—Includes a human-readable description of the error
* locations—Specifies where in the query the error was found, including the line and column

Listing 9.4 shows what happens when you run the query with empty parentheses. As you can see, we get a syntax error with a somewhat cryptic message: Expected Name, found ). This is a common error that occurs whenever you make a syntax error in GraphQL. In this case, it means that GraphQL was expecting a parameter after the opening parenthesis, but instead it found a closing parenthesis.

Listing 9.4 Missing query parameter errors

# Query:

{

ingredient() { ①

name

}

}

# Error:

{

"errors": [ ②

{

"message": "Syntax Error: Expected Name, found )", ③

"locations": [ ④

{

"line": 2, ⑤

"column": 14 ⑥

}

]

}

]

}

① We run the ingredient() query without the required parameter id.

② An unsuccessful response includes an "errors" key.

③ We get a generic syntax error.

④ The precise location of the error in our query

⑤ The error was found in the second line of our query document.

⑥ The error was found at the 14th character in the second line.

On the other hand, if you run the ingredient() query without any parentheses at all, as shown in listing 9.5, you’ll get an error specifying that you missed the required parameter id.

**USE OF PARENTHESES IN GRAPHQL QUERIES AND MUTATIONS** In GraphQL, the parameters of a query are defined within parentheses. If you run a query with required parameters, such as ingredient, you must include the parameters within parentheses (see listing 9.3). Failing to do so will throw an error (see listings 9.4 and 9.5). If you run a query without parameters, you must omit the parentheses. For example, when we run the allIngredients() query, we omit parentheses (see listing 9.1), since allIngredients() doesn’t require any parentheses.

Listing 9.5 Missing query parameter errors

# Query:

{

ingredient { ①

name

}

}

# Error:

{

"errors": [

{

"message": "Field \"ingredient\" argument \"id\" of type \"ID!\" is

➥ required, but it was not provided.", ②

"locations": [

{

"line": 2, ③

"column": 3 ④

}

]

}

]

}

① We run the ingredient() query without the parentheses.

② The error message says that the id parameter is missing in the query.

③ The error was found in the second line of our query document.

④ The error was found at the third character of the second line.

Now that we know how to read and interpret error messages when we make mistakes in our queries, let’s explore queries that return multiple types.

**9.3 Using fragments in queries**

This section explains how we run queries that return multiple types. The queries that we’ve seen so far in this chapter are simple since they only return one type, which is Ingredient. However, our product-related queries, such as allProducts() and product(), return the Product union type, which is the combination of the Cake and Beverage types. How do we run our queries in this case?

When a GraphQL query returns multiple types, we must create selection sets for each type. For example, if you run the allProducts() query with a single selector, you get the error message saying that the server doesn’t know how to resolve the properties in the selection set.

Listing 9.6 Calling allProducts() with a single selector set

# Query

{

allProducts { ①

name ②

}

}

# Error message

{

"errors": [ ③

{

"message": "Cannot query field \"name\" on type \"Product\". Did you

➥ mean to use an inline fragment on \"ProductInterface\", \"Beverage\",

➥ or \"Cake\"?", ④

"locations": [

{

"line": 3, ⑤

"column": 5 ⑥

}

]

}

]

}

① We run the allProducts() query without parameters.

② We include the name property in the selection set.

③ We get an error response.

④ The server doesn’t know how to resolve the properties in the selection set.

⑤ The error was found in the third line of the query document.

⑥ The error was found at the fifth position of the third line.

The error message in listing 9.6 asks you whether you meant to use an inline fragment on either ProductInterface, Beverage, or Cake. What is an inline fragment? An *inline fragment* is an anonymous selection set on a specific type. The syntax for inline fragments includes three dots (the spread operator in JavaScript) followed by the on keyword and the type on which the selection set applies, as well as a selection of properties between curly braces:

...on ProductInterface {

name

}

Listing 9.7 fixes the allProducts() query by adding inline fragments that select properties on the ProductInterface, Cake, and Beverage types. allProducts()’s return type is Product, which is the union of Cake and Beverage, so we can select properties from both types. From the specification, we also know that Cake and Beverage implement the ProductInterface interface type, so we can conveniently select properties common to both Cake and Beverage directly on the interface.

Listing 9.7 Adding inline fragments for each return type

{

allProducts {

...on ProductInterface { ①

name

}

...on Cake { ②

hasFilling

}

...on Beverage { ③

hasCreamOnTopOption

}

}

}

① Inline fragment with a selection set on the ProductInterface type

② Inline fragment with a selection set on the Cake type

③ Inline fragment with a selection set on the Beverage type

Listing 9.7 uses inline fragments, but the real benefit of fragments is we can define them as standalone variables. This makes fragments reusable, and it also makes our queries more readable. Listing 9.8 shows how we can refactor listing 9.7 to use standalone fragments. The queries are so much cleaner! In real-life situations, you’re likely to work with large selection sets, so organizing your fragments into standalone, reusable pieces of code will make your queries easier to read.

Listing 9.8 Using standalone fragments

{

allProducts {

...commonProperties

...cakeProperties

...beverageProperties

}

}

fragment commonProperties on ProductInterface {

name

}

fragment cakeProperties on Cake {

hasFilling

}

fragment beverageProperties on Beverage {

hasCreamOnTopOption

}

Now that we know how to deal with queries that return multiple object types, let’s take our querying skills to the next level. In the next section, we’ll learn to run queries with a specific type of parameter called an input parameter.

**9.4 Running queries with input parameters**

This section explains how we run queries with input type parameters. In section 8.8, we learned that input types are similar to object types, but they’re meant for use as parameters for a GraphQL query or mutation. One example of an input type in the products API is ProductsFilter, which allows us to filter products by factors such as availability, minimum or maximum price, and others. ProductsFilter is the parameter of the products() query. How do we call the products() query?

When a query takes parameters in the form of an input type, the query’s input type parameter must be passed in the form of an input object. This may sound complicated, but it’s actually very simple. We call the products() query using ProductsFilter’s maxPrice parameter. To use any of the parameters in the input type, we simply wrap them with curly braces.

Listing 9.9 Calling a query with a required parameter

{

products(input: {maxPrice: 10}) { ①

...on ProductInterface { ②

name

}

}

}

① We specify ProductFilter’s maxPrice parameter.

② Inline fragment on the ProductInterface type

Now that we know how to call queries with input parameters, let’s take a deeper look at the relationships between the objects defined in the API specification and see how we can build queries that allow us to traverse our data graph.

**9.5 Navigating the API graph**

This section explains how we select properties from multiple types by leveraging their connections. In section 8.5, we learned to create connections between object types by using edge properties and through types. These connections allow API clients to traverse the graph of relationships between the resources managed by the API. For example, in the products API, the Cake and Beverage types are connected with the Ingredient type by means of a through type called IngredientRecipe. By leveraging this connection, we can run queries that fetch information about the ingredients related to each product. In this section, we’ll learn to build such queries.

In our queries, whenever we add a selector for a property that points to another object type, we must include a nested selection set for said object type. For example, if we add a selector for the ingredient property on the ProductInterface type, we have to include a selection set with any of the properties in IngredientRecipe nested within the ingredients property. We include a nested selection set for the ingredients property of ProductInterface in the allProducts() query. The query selects the name of each product as well as the name of each ingredient in the product’s recipe.

Listing 9.10 Querying nested object types

{

allProducts {

...on ProductInterface { ①

name,

ingredients { ②

ingredient { ③

name

}

}

}

}

}

① Inline fragment on the ProductInterface type

② Selector for ProductInterface’s ingredients property

③ Selector for IngredientRecipe’s ingredient property

Listing 9.10 leverages the connection between the ProductInterface and Ingredient types to fetch information from both types in a single query, but we can take this further. The Ingredient type contains a supplier property, which points to the Supplier type. Say we want to get a list of products, including their names and ingredients, together with the supplier’s name of each ingredient. (I encourage you to head over to the Voyager UI generated by graphql-faker to visualize the relationships captured by this query; figure 9.1 is an illustration of the Voyager UI.)

Listing 9.11 Traversing the products API graph through connections between types

{

allProducts {

...on ProductInterface { ①

name

ingredients { ②

ingredient { ③

name

supplier { ④

name

}

}

}

}

}

}

① Inline fragment on the ProudctInterface type

② Selector for ProductInterface’s ingredients property

③ Selector for IngredientRecipe’s ingredient property

④ Selector for Ingredient’s supplier property

Listing 9.11 is traversing our graph of types. Starting from the ProductInterface type, we are able to fetch details about other objects, such as Ingredient and Supplier, by leveraging their connections.

Here lies one the most powerful features of GraphQL, and one of its main advantages in comparison with other types of APIs, such as REST. Using REST, we’d need to make multiple requests to obtain all the information we were able to fetch in one request in listing 9.11. GraphQL gives you the power to obtain all the information you need, and just the information you need, in a single request.

Now that we know how to traverse the graph of types in a GraphQL API, let’s take our querying skills to the next level by learning how to run multiple queries within a single request!

**9.6 Running multiple queries and query aliasing**

This section explains how to run multiple queries per request and how to create aliases for the responses returned by the server. Aliasing our queries means changing the key under which the dataset returned by the server is indexed. As we’ll see, aliases can improve the readability of the results returned by the server, especially when we make multiple queries per request.

**9.6.1 Running multiple queries in the same request**

In previous sections, we ran only one query per request. However, GraphQL also allows us to send several queries in one request. This is yet another powerful feature of GraphQL that can help us save unnecessary network round-trips to the server, improving the overall performance of our applications and therefore user experience.

Let’s say we wanted to obtain a list of all the products and ingredients available in the CoffeeMesh platform, as shown in figure 9.4. To do that, we can run allIngredients() with the allProducts() queries. Listing 9.12 shows how we include both operations within the same query document. By including multiple queries within the same query document, we make sure all of them are sent to the server in the same request, and therefore we save round-trips to the server. The code also includes a named fragment that selects properties on the ProductInterface type. Named fragments are useful to keep our queries clean and focused.

Listing 9.12 Multiple queries per request

{

allProducts { ①

...commonProperties ②

}

allIngredients { ③

name

}

}

fragment commonProperties on ProductInterface { ④

name

}

① We run the allProducts() query without parameters.

② We select properties using a named fragment.

③ We run the allIngredients() query.

④ Named fragment with selection set on the ProductInterface type

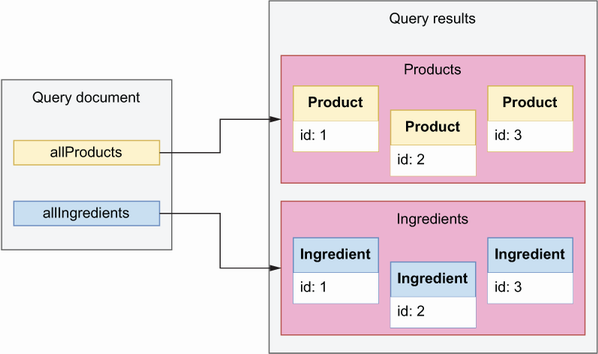


Figure 9.4 In GraphQL, we can run multiple queries within the same request, and the response will contain one dataset for each query.

**9.6.2 Aliasing our queries**

All the queries we’ve run in previous sections are anonymous queries. When we make an anonymous query, the data returned by the server appears under a key named after the name of the query we’re calling.

Listing 9.13 Result of an anonymous query

# Query:

{

allIngredients { ①

name

}

}

# Result:

{

"data": { ②

"allIngredients": [ ③

{

"name": "string"

},

{

"name": "string"

}

]

}

}

① We run the allIngredients() query.

② Successful response from the query

③ Query result

Running anonymous queries can sometimes be confusing. allIngredients() returns a list of ingredients, so it is helpful to index the list of ingredients under an ingredients key, instead of allIngredients(). Changing the name of this key is called *query aliasing*. We can make our queries more readable by using aliasing. The benefits of aliasing become clearer when we include multiple queries in the same request. For example, the query for all products and ingredients shown in listing 9.12 becomes more readable if we use aliases. The following code shows how we use aliases to rename the results of each query: the result of allProducts() appears under the product alias, and the result of the allIngredients() query appears under the ingredients alias.

Listing 9.14 Using query aliasing for more readable queries

{

products: allProducts { ①

...commonProperties ②

}

ingredients: allIngredients { ③

name

}

}

fragment commonProperties on ProductInterface { ④

name

}

① Alias for the allProducts() query

② We select properties using a named fragment.

③ Alias for the allIngredients() query

④ Named fragment with selection set on the ProductInterface

In some cases, using query aliases is necessary to make our requests work. For example, in listing 9.15, we run the products() query twice to select two datasets: one for available products and another for unavailable products. Both datasets are produced by the same query: products. As you can see, without query aliasing, this request results in conflict error, because both datasets return under the same key: products.

Listing 9.15 Error due to calling the same query multiple times without aliases

{

products(input: {available: true}) { ①

...commonProperties ②

}

products(input: {available: false}) { ③

...commonProperties

}

}

fragment commonProperties on ProductInterface { ④

name

}

# Error

{

"errors": [ ⑤

{

"message": "Fields \"products\" conflict because they have differing

➥ arguments. Use different aliases on the fields to fetch both if this

➥ was intentional.", ⑥

"locations": [

{

"line": 2, ⑦

"column": 3

},

{

"line": 5,

"column": 3

}

]

}

]

}

① We run the products() query filtering for available products.

② We select properties using the commonProperties fragment.

③ We run the products() query filtering for unavailable products.

④ Named fragment with selection set on the ProductInterface type.

⑤ The query returns an unsuccessful response, so the payload includes an error key.

⑥ The error message says that the query document contains a conflict.

⑦ The server found errors in lines 2 and 5 from the query document.

To resolve the conflict created by the queries in listing 9.15, we must use aliases. Listing 9.16 fixes the query by adding an alias to each operation: availableProducts for the query that filters for available products and unavailableProducts for the query that filters for unavailable products.

Listing 9.16 Calling the same query multiple times with aliases

{

availableProducts: products(input: {available: true}) { ①

...commonProperties

}

unavailableProducts: products(input: {available: false}) { ②

...commonProperties

}

}

fragment commonProperties on ProductInterface {

name

}

# Result (datasets omitted for brevity)

{

"data": { ③

"availableProducts": [...], ④

"unavailableProducts": [...] ⑤

}

}

① Alias for the available products() query

② unavailableProducts alias for the unavailable products() query

③ Successful response from the server

④ Result of the available products() query

⑤ Result of the unavailable products() query

This concludes our overview of GraphQL queries. You’ve learned to run queries with parameters, with input types, with inline and named fragments, and with aliases, and you’ve learned to include multiple queries within the same request. We’ve come a long way! But no overview of the GraphQL query language would be complete without learning how to run mutations.

**9.7 Running GraphQL mutations**

This section explains how we run GraphQL mutations. Mutations are GraphQL functions that allow us to create resources or change the state of the server. Running a mutation is similar to running a query. The only difference between the two is in their intent: queries are meant to read data from the server, while mutations are meant to create or change data in the server.

Let’s illustrate how we run a mutation with an example. Listing 9.17 shows how we run the deleteProduct() mutation. When we use mutations, we must start our query document by qualifying our operation as a mutation. The deleteProduct() mutation has one required argument, a product ID, and its return value is a simple Boolean, so in this case, we don’t have to include a selection set.

Listing 9.17 Calling a mutation

mutation { ①

deleteProduct(id: "asdf") ②

}

① We qualify the operation we’re going to run as a mutation.

② We call the deleteProduct() mutation, passing in the required id parameter.

Let’s now look at a more complex mutation, like addProduct(), which is used to add new products to the CoffeeMesh catalogue. addProduct() has three required parameters:

* name—The product name.
* type—The product type. The values for this parameter are constrained by the ProductType enumeration, which offers two choices: cake and beverage.
* input—Additional product properties, such as its price, size, list of ingredients, and others. The full list of properties is given by the AddProductInput type.

addProduct() returns a value of type Product, which means, in this case, we must include a selection set. Remember that Product is the union of the Cake and Beverage types, so our selection set must use fragments to indicate which type’s properties we want to include in our return payload. In the following example, we select the name property on the ProductInterface type.

Listing 9.18 Calling a mutation with input parameters and complex return type

mutation { ①

addProduct(name: "Mocha", type: beverage, input: {price: 10, size: BIG, ingredients: [{ingredient: 1, quantity: 1, unit: LITERS}]}) { ②

...commonProperties ③

}

}

fragment commonProperties on ProductInterface {

name

}

① We qualify the operation we’re going to run as a mutation.

② We call the addProduct() mutation.

③ We select properties using a named fragment.

Now that we know how to run mutations, it’s time to learn how we write more structured and readable query documents by parameterizing the arguments.

**9.8 Running parameterized queries and mutations**

This section introduces parameterized queries and explains how we can use them to build more structured and readable query documents. In previous sections, when using queries and mutations that require parameters, we defined the values for each parameter in the same line we called the function. In queries with lots of arguments, this approach can lead to query documents, which are cluttered and difficult to read and maintain. GraphQL offers a solution for this, which is to use parameterized queries.

Parameterized queries allow us to decouple our query/mutation calls from the data. Figure 9.5 illustrates how we parameterize the call to the addProduct() mutation using GraphiQL (the code for the query is also shown in listing 9.19 so that you can inspect it and copy it more easily). There’re two things we need to do when we parameterize a query or mutation: create a function wrapper around the query/mutation, and assign values for the query/mutation parameters in a query variables object. Figure 9.6 illustrates how all these pieces fit together to bind the parameterized values to the addProduct() mutation call.

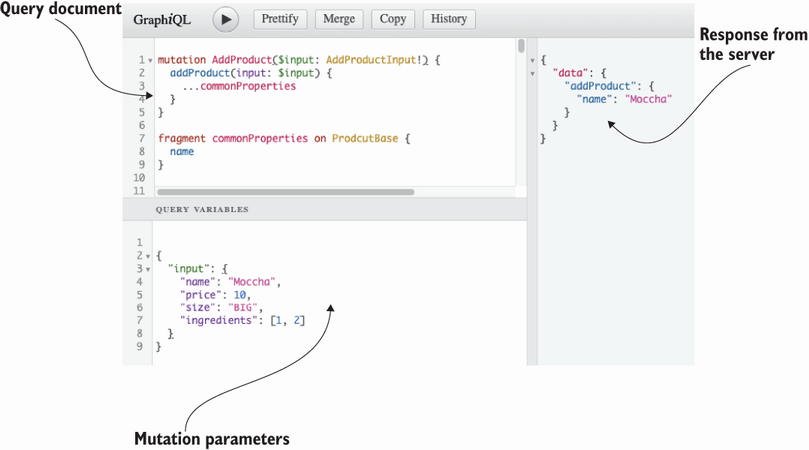


Figure 9.5 GraphiQL offers a Query Variables panel where we can include the input values for our parameterized queries.

Listing 9.19 Using parameterized syntax

# Query document

mutation CreateProduct( ①

$name: String!

$type: ProductType!

$input: AddProductInput!

) {

addProduct(name: $name, type: $type, input: $input) { ②

...commonProperties ③

}

}

fragment commonProperties on ProductInterface {

name

}

# Query variables

{

"name": "Mocha", ④

"type": "beverage", ⑤

"input": { ⑥

"price": 10,

"size": "BIG",

"ingredients": [{"ingredient": 1, "quantity": 1, "unit": "LITERS"}]

}

}

① We create a wrapper named CreateProduct().

② We call the addProduct() mutation.

③ We select properties using a named fragment.

④ We assign a value to the name parameter.

⑤ We assign a value to the type parameter.

⑥ We assign a value to the input parameter.

Let’s look at each of these steps in detail.

1. *Creating a query/mutation wrapper*. To parameterize our queries, we create a function wrapper around the query or mutation. In figure 9.5, we call the wrapper CreateProduct(). The syntax for the wrapper looks very similar to the syntax we use to define a query. Parameterized arguments must be included in the wrapper’s function signature. In figure 9.5, we parameterize the name, type, and input parameters of the addProduct() mutation. The parameterized argument is marked with a dollar sign ($). In the wrapper’s signature (i.e., in CreateProduct()), we specify the expected type of the parameterized arguments.
2. *Parameterizing through a query variables object*. Separately, we define our query variables as a JSON document. As you can see in figure 9.5, in GraphiQL we define query variables within the Query Variables panel. For further clarification on how parameterized queries work, look at figure 9.6.

In figure 9.5, we used parameterized syntax to wrap only one mutation, but nothing prevents us from wrapping more mutations within the same query document. When we wrap multiple queries or mutations, all the parameterized arguments must be defined within the wrapper’s function signature. The following code shows how we extend the query from listing 9.19 to include a call to the deleteProduct() mutation. Here, we call the wrapper CreateAndDeleteProduct() to better represent the actions in this request.

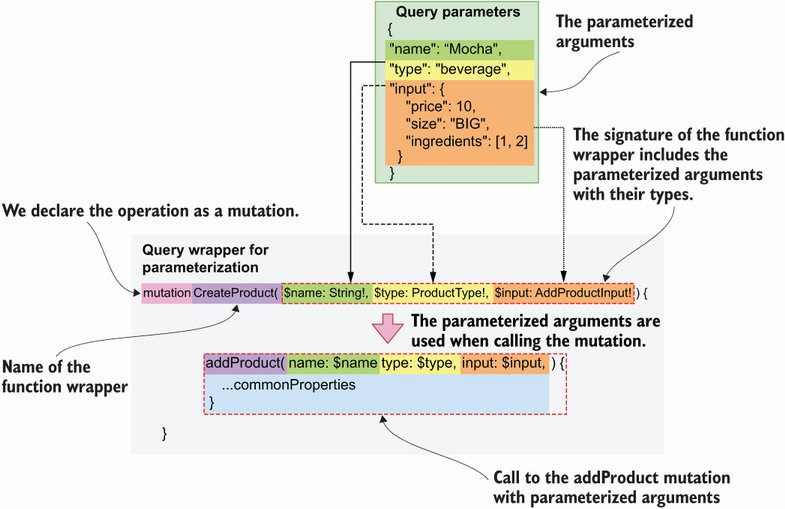


Figure 9.6 To parameterize queries and mutations, we create a function wrapper around the query or mutation. In the wrapper’s signature we include the parameterized arguments. Parameterized variables carry a leading dollar ($) sign.

Listing 9.20 Using parameterized syntax

# Query document

mutation CreateAndDeleteProduct( ①

$name: String!

$type: ProductType!

$input: AddProductInput!

$id: ID!

) {

addProduct(name: $name, type: $type, input: $input) { ②

...commonProperties ③

}

deleteProduct(id: $id) ④

}

fragment commonProperties on ProductInterface {

name

}

# Query variables

{

"name": "Mocha", ⑤

"type": "beverage",

"input": {

"price": 10,

"size": "BIG",

"ingredients": [{"ingredient": 1, "quantity": 1, "unit": "LITERS"}]

},

"id": "asdf" ⑥

}

① We created a wrapper named CreateAndDeleteProduct().

② We call the addProduct() mutation.

③ We select properties using a named fragment.

④ We call the deleteProduct() mutation.

⑤ We assign values to addProduct()’s parameters.

⑥ We set the value for deleteProduct()’s id parameter.

This completes our journey through learning how to consume GraphQL APIs. You can now inspect any GraphQL API, explore its types, and play around with its queries and mutations. Before we close this chapter, I’d like to show you how a GraphQL API request works under the hood.

**9.9 Demystifying GraphQL queries**

This section explains how GraphQL queries work under the hood in the context of HTTP requests. In previous sections, we used the GraphiQL client to explore our GraphQL API and to interact with it. GraphiQL translates our query documents into HTTP requests that the GraphQL server understands. GraphQL clients such as GraphiQL are interfaces that make it easier to interact with a GraphQL API. But nothing prevents you from sending an HTTP request directly to the API, say, from your terminal, using something like cURL. Contrary to a popular misconception, you don’t really need any special tools to work with GraphQL APIs.[**1**](https://learning.oreilly.com/library/view/microservice-apis/9781617298417/OEBPS/Text/09.htm#pgfId-1098532)

To send a request to a GraphQL API, you can use either of the GET or POST methods. If you use GET, you send your query document using URL query parameters, and if you use POST, you include the query in the request payload. GraphQL Faker’s mock server only accepts GET requests, so I’ll illustrate how you send a query using GET.

Let’s run the allIngredients() query, selecting only the name property of each ingredient. Since this is a GET request, our query document must be included in the URL as a query parameter. However, the query document contains special characters, such as curly braces, which are considered unsafe and therefore cannot be included in a URL. To deal with special characters in URLs, we URL encode them. *URL encoding* is the process of translating special characters, such as braces, punctuation marks, and others, into a suitable format for URLs. URL-encoded characters start with a percent sign, so this type of encoding is also known as *percent encoding*.[**2**](https://learning.oreilly.com/library/view/microservice-apis/9781617298417/OEBPS/Text/09.htm#pgfId-1098539) cURL takes care of URL encoding our data when we use the --data-urlencode option. By using --data-urlencode, cURL translates our command into a GET request with the following URL: http://localhost:9002/graphql?query=%7BallIngredients%7Bname%7D%7D. The following snippet shows the cURL command you need to run to make this call:

$ curl http://localhost:9002/graphql --data-urlencode \

'query={allIngredients{name}}'

Now that you understand how GraphQL API requests work under the hood, let’s see how we can leverage this knowledge to write code in Python that consumes a GraphQL API.

**9.10 Calling a GraphQL API with Python code**

This section illustrates how we can interact with a GraphQL API using Python. GraphQL clients like GraphiQL are useful to explore and get familiar with a GraphQL API, but in practice, you’ll spend most of your time writing applications that consume those APIs programmatically. In this section, we learn to consume the products API using a GraphQL client written in Python.

To work with GraphQL APIs, the Python ecosystem offers libraries such as gql (<https://github.com/graphql-python/gql>) and sgqlc (<https://github.com/profusion/sgqlc>). These libraries are useful when we want to use advanced features of GraphQL, such as subscriptions. You’ll rarely need those features in the context of microservices, so for the purposes of this section, we’ll take a simpler approach and use the popular requests library (<https://github.com/psf/requests>). Remember that GraphQL queries are simply GET or POST requests with a query document.

Listing 9.21 shows how we call the allIngredients() query, adding a selector for Ingredient’s name property. The listing is also available under ch09/client.py in this book’s GitHub repository. Since our GraphQL mock server only accepts GET requests, we send the query document in the form of URL-encoded data. With requests, we accomplish this by passing the query document through the get method’s params argument. As you can see, the query document looks the same as what we wrote in the GraphiQL query panel, and the result from the API also looks the same. This is great news, because it means that, when working out your queries, you can start working with GraphiQL, leveraging its great support for syntax highlighting and query validation, and when you’re ready, you can move your queries directly to your Python code.

Listing 9.21 Calling a GraphQL query using Python

# file: ch09/client.py

import requests ①

URL = 'http://localhost:9002/graphql' ②

query\_document = ''' ③

{

allIngredients {

name

}

}

'''

result = requests.get(URL, params={'query': query\_document}) ④

print(result.json()) ⑤

# Result

{'data': {'allIngredients': [{'name': 'string'}, {'name': 'string'},

➥ {'name': 'string'}]}}

① We import the requests library.

② The base URL of our GraphQL server

③ The query document

④ We send a GET request to the server with the query document as a URL query parameter.

⑤ We parse and print the JSON payload returned by the server.

This concludes our journey through GraphQL. You went from learning about the basic scalar types supported by GraphQL in chapter 8 to making complex queries using tools as varied as GraphiQL, cURL, and Python in this chapter. Along the way, we built the specification for the products API, and we interacted with it using a GraphQL mock server. That’s no small feat. If you’ve read this far, you’ve learned a great deal of things about APIs, and you should be proud of it!

GraphQL is one of the most popular protocols in the world of web APIs, and its adoption grows every year. GraphQL is a great choice for building microservices APIs and for integration with frontend applications. In the next chapter, we’ll undertake the actual implementation of the products API and its service. Stay tuned!