CHAPTER 24

Understanding GraphQL

GraphQL is an end-to-end system for creating and consuming APIs, providing a more flexible alternative to using traditional RESTful web services, such as the one created in Chapter 23. In this chapter, I explain how GraphQL services are defined and how queries are performed. In Chapter 25, I demonstrate the different ways that a GraphQL API can be consumed by a React application. Table 24-1 puts GraphQL in context.

Table 24-1. *Putting GraphQL in Context*

Question	Answer
What is it?	GraphQL is a query language that produces APIs.
Why is it useful?	GraphQL provides the client with flexible access to data, ensuring that the client receives only the data it requires and allowing new queries to be formulated without requiring server-side changes.
How is it used?	At the server, a schema is defined and implemented using resolver functions. The client uses the GraphQL language to send queries and request changes.
Are there any pitfalls or limitations?	GraphQL is complex and writing a useful schema can require skill.
Are there any alternatives?	Clients can use RESTful web services, as described in Chapter 23.

Note I describe the features of GraphQL that are most useful for React development. For a complete description of GraphQL, see the GraphQL specification at `https://facebook.github.io/graphql/June2018`.

A. Freeman, *Pro React 16*, https://doi.org/10.1007/978-1-4842-4451-7_24

Table 24-2 summarizes the chapter.

Table 24-2. *Chapter Summary*

Problem	Solution	Listing
Define a GraphQL service	Describe the queries and mutations that will be supported and implement the resolvers that provide them	3, 4, 8–10, 20–21
Query a GraphQL service	Specify the query name and the fields that are required in the result	7, 11, 27, 28
Filter results	Specify query arguments	12–19
Make changes using a GraphQL service	Specify the mutation and the fields for the update	22–24
Parameterize queries	Use query variables	25, 26
Request the same set of fields from multiple queries	use a query fragment	29

Preparing for This Chapter

In this chapter, I continue to use the example application from Chapter 23. To prepare for this chapter, open a command prompt, navigate to the `productapp` folder, and run the commands shown in Listing 24-1 to add packages to the project.

■ **Tip** You can download the example project for this chapter—and for all the other chapters in this book—from https://github.com/Apress/pro-react-16.

Listing 24-1. Adding Packages

```
npm install --save-dev graphql@14.0.2
npm install --save-dev express@4.16.4
npm install --save-dev express-graphql@0.7.1
npm install --save-dev graphql-import@0.7.1
npm install --save-dev cors@2.8.5
```

For quick reference, the packages by the commands in Listing 24-1 are described in Table 24-3.

Table 24-3. *The Packages Added to the Project*

Name	Description
`graphql`	This package contains the reference implementation of GraphQL.
`express`	This package provides an extensible HTTP server and will be the foundation of the GraphQL server used in this chapter.
`express-graphql`	This package provides GraphQL services over HTTP through the `express` package.
`graphql-import`	This package allows GraphQL schemas to be defined in multiple files and imports schemas more easily than reading a file directly.
`cors`	This package enables Cross-Origin Resource Sharing (CORS) for the Express HTTP server.

Once the packages are installed, use the command prompt to run the command shown in Listing 24-2 in the `productapp` folder to start the development tools. The RESTful web service defined in Chapter 23 is also started and is still used by the application.

Listing 24-2. Starting the Development Tools

```
npm start
```

Once the initial preparation for the project is complete, a new browser window will open and display the URL `http://localhost:3000`, as shown in Figure 24-1.

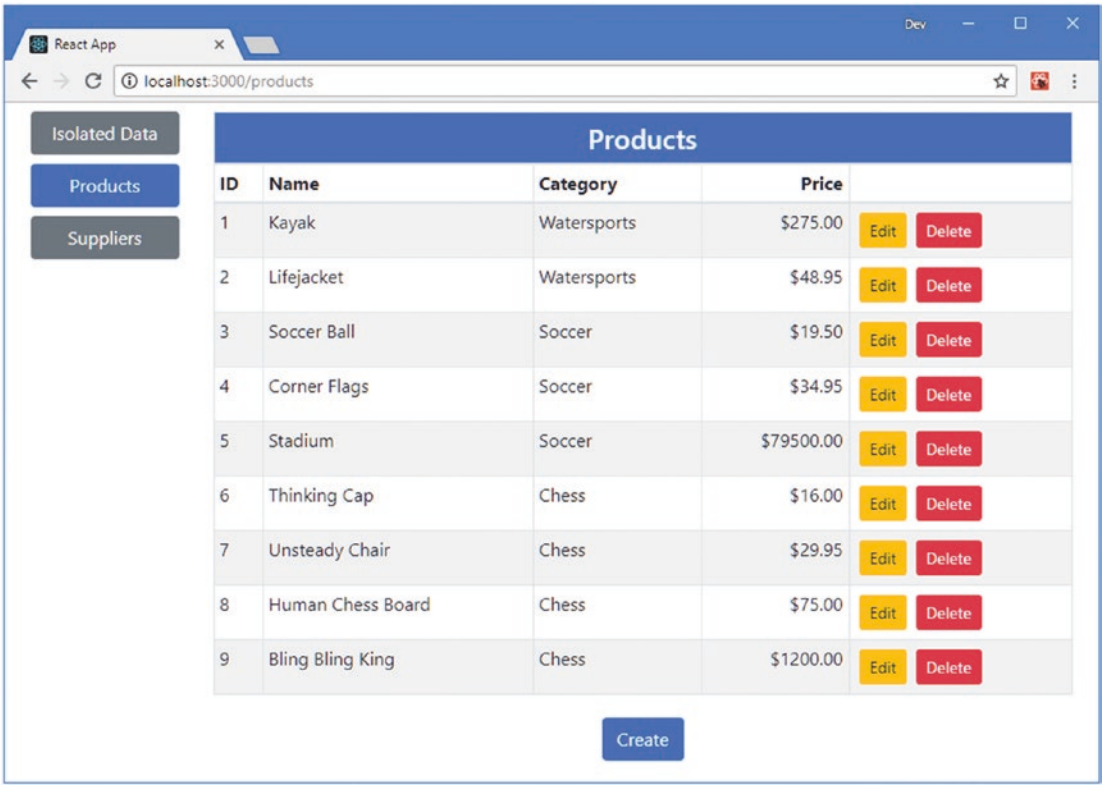

Figure 24-1. *Running the example application*

Understanding GraphQL

RESTful web services are easy to get started with, but they can become inflexible as the needs of the client evolve and the number of client applications using the service increases.

Changes that suit one application can't be made because they cause problems in another application, work backs up so that changes are not made in time for client-application release dates, and infrastructure development teams struggle to balance competing demands for features. You may not have any avenue for requesting changes when you rely on a third-party web service because you are one of dozens or hundreds of development teams all asking for new features.

The result is a poor fit between the application and the web service it relies on. Clients often have to make multiple requests to the web service to get the data they require; this is then merged into a useful format. The client has to understand how the objects returned from different REST requests relate to one another and often has to request data that is then discarded because only a subset of the data fields is required.

The underlying problem with REST web services is that the data they provide and the way they provide it are fixed, and that becomes an issue as the needs of the client application change. GraphQL addresses this problem by allowing the client more control over what data is requested and how it is expressed, with the result that client-side applications can add features that use data in new ways with fewer server-side changes required.

UNDERSTANDING THE DRAWBACKS OF GRAPHQL

GraphQL isn't suitable for every situation. GraphQL is complex and not as widely understood as REST, which can make it difficult to find experienced developers and robust and well-tested tools and libraries. And GraphQL can shift work that would have been performed by the client application into the server, which can increase costs in the data center and require licenses for back-end servers that support GraphQL.

It is important to consider GraphQL as an option, especially if your application is likely to require ongoing development after deployment or you intend to develop or support multiple client applications. But my advice is not to rush into using GraphQL until you are sure that a REST web service won't give you the flexibility you need.

Creating the GraphQL Server

I am going to create a custom GraphQL server that presents the same data as the web service from Chapter 23 The process of creating the GraphQL service isn't required by all projects, especially when consuming a third-party API, but understanding what is happening at the server provides useful insights into how GraphQL works. In the sections that follow, I'll go through the process of describing the types of requests that the client will be able to make and write the code required to deal with those requests.

CHOOSING AN ALTERNATIVE GRAPHQL SERVER

I use the GraphQL reference implementations to create a simple GraphQL server for this chapter. It makes it easy to demonstrate how GraphQL works but doesn't make any provision for working with real data.

For small and simple projects, adding persistent data support with a package such as Lowdb (`https://github.com/typicode/lowdb`) or MongoDB (`https://www.mongodb.com`) may be suitable.

For more complex projects, the Apollo server (`https://github.com/apollographql/apollo-server`) is the most common choice. There are open-source and paid-for plans available, and there is a wide range of data integration options available, such as using GraphQL as a front end to existing REST web services.

Creating the Schema

GraphQL describes the requests that can be performed using a schema, which is written in the GraphQL schema language. I created the `src/graphql` folder and added to it a file called `schema.graphql` with the contents shown in Listing 24-3.

Listing 24-3. The Contents of the schema.graphql File in the src/graphql Folder

```
type product {
    id: ID!,
    name: String!,
    category: String!
    price: Float!
}

type supplier {
    id: ID!,
    name: String!,
    city: String!,
    products: [ID]
}

type Query {
    products: [product],
    suppliers: [supplier]
}
```

The schema defined in Listing 24-3 defines two custom types: `product` and `supplier`. These types will be used as the results of the queries supported by the GraphQL server. Each result type is defined with a set of fields, each of which is typed like this:

```
...
category: String!
...
```

The name of this field is `category`, and its type is `String`. GraphQL provides a set of built-in types, which are described in Table 24-4. The exclamation mark (the `!` character) after the field type indicates that values for this field are mandatory. Fields can also return arrays of values, like this:

```
...
products: [ID]
...
```

Table 24-4. *The Built-in GraphQL Types*

Name	Description
`ID`	This type represents a unique identifier.
`String`	This type represents a string.
`Int`	This type represents a signed integer
`Float`	This type represents a floating-point value
`Boolean`	This type represents a `true` or `false` value.

The square brackets indicate that the `products` field of the `supplier` type will be an array of `ID` values.

■ **Tip** Don't worry about the GraphQL type system too much at the moment. It will start to make more sense as you see how the different parts of the server fit together and are used by the client.

In addition to the built-in types, GraphQL supports the `Query` type, which is used to define the queries that the server will support. There are two queries defined in the schema in Listing 24-3.

```
...
type Query {
    products: [product],
    suppliers: [supplier]
}
...
```

The first statement defines a query called `products` that will return an array of `product` objects. The second statement defines a query called `suppliers` that will return an array of `supplier` objects.

Creating the Resolvers

The next step is to write the functions that implement the `products` and `suppliers` queries defined in Listing 24-3. I added a file called `resolvers.js` to the `src/graphql` folder, with the code shown in Listing 24-4.

Listing 24-4. The Contents of the resolvers.js File in the src/graphql Folder

```
var data = require("../../restData")();

module.exports = {

    products: () => data.products,
    suppliers: () => data.suppliers
}
```

Each resolver is a function whose name corresponds to one of the queries and that returns data in the format declared by the schema. The data used by the `products` and `suppliers` resolvers uses data loaded from the `restData.js` file.

■ **Note** The GraphQL server will be run by Node.js, which does not support JavaScript modules at the time of writing, which means that the `import` and `export` keywords cannot be used. Instead, the `require` function is used to declare a dependency on a file, and `module.exports` is used to make code or data available outside of a JavaScript file.

Creating the Server

The final step is to create the code that will process the schema and the resolvers and create the GraphQL server. I added a file called `graphqlServer.js` in the `productapp` folder and added the code shown in Listing 24-5.

Listing 24-5. The Contents of the graphqlServer.js File in the productapp Folder

```
var { buildSchema } = require("graphql");
var { importSchema } = require("graphql-import");
var express = require("express");
var graphqlHTTP = require("express-graphql")
var cors = require("cors")
var schema = importSchema("./src/graphql/schema.graphql");
var resolvers = require("./src/graphql/resolvers");

var app = express();

app.use(cors());
app.use("/graphql", graphqlHTTP({
  schema: buildSchema(schema),
  rootValue: resolvers,
  graphiql: true,
}));
app.listen(3600, () => console.log("GraphQL Server Running on Port 3600"));
```

The `graphql` package provides the `buildSchema` function, which takes a schema string and prepares it for use. The contents of the schema file are imported using the `graphql-import` package and passed to the `buildSchema` function. The `express-graphql` package integrates GraphQL support into the popular `express` server, which I have configured to listen on port 3600.

To start the GraphQL server, open a new command prompt, navigate to the `productapp` folder, and run the command shown in Listing 24-6. (The GraphQL server won't automatically reload when there are schema or resolver changes and will have to be restarted for some of the examples in this chapter, which is why I have not integrated it into the `npm start` command, as I did for the RESTful web service.)

Listing 24-6. Starting the GraphQL Server

```
node graphqlServer.js
```

The GraphQL server includes support for GraphiQL (pronounced *graphical*), which is a browser-based GraphQL tool. To make sure that the GraphQL server is working, open a new browser tab and navigate to `http://localhost:3600/graphql`, which should show the tool in Figure 24-2.

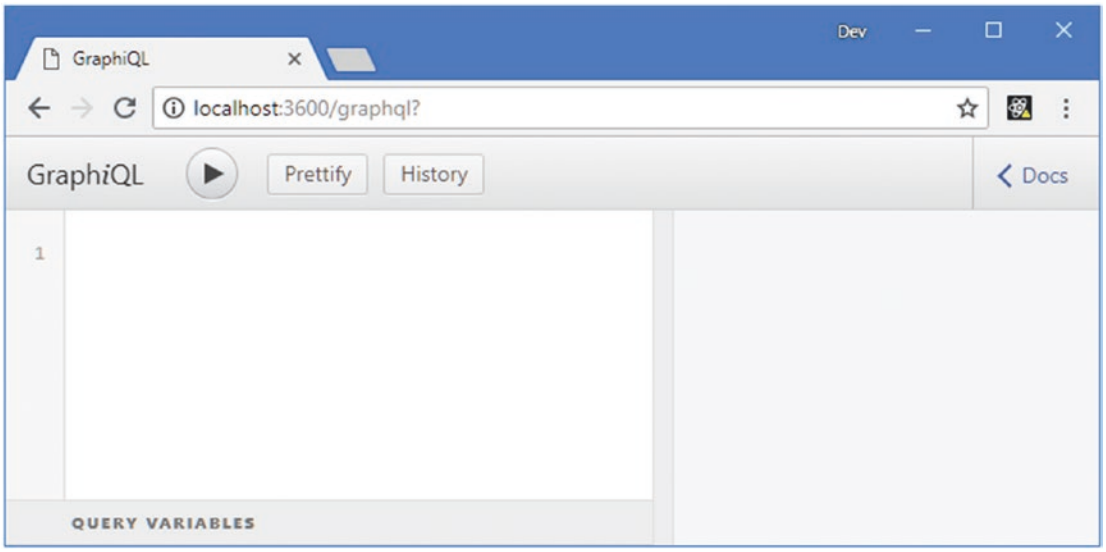

Figure 24-2. *The GraphiQL browser*

Making GraphQL Queries

The GraphiQL tool makes it easy to perform queries before integrating GraphQL into the example application. To query for all the supplier objects, for example, enter the query shown in Listing 24-7 into the left pane of the GraphiQL window.

Listing 24-7. A Query for Supplier Data

```
query {
  suppliers {
    id,
    name,
    city,
    products
  }
}
```

The query is basic, but it reveals a lot about how GraphQL queries work. The `query` keyword is used to differentiate between requests to retrieve data and *mutations*, which are used to make changes (and which are described in the "Making GraphQL Mutations" section). The query itself is enclosed in curly brackets, also known as *braces*. Inside the braces, the query name is specified, which is `suppliers` in this case.

When you query a GraphQL service, you must specify the data fields that you want to receive. Unlike a REST web service, which always presents the same data structures, GraphQL allows the client to specify the results it wants to receive, enclosed in another set of braces. The query in Listing 24-7 selects the `id`, `name`, `city`, and `products` fields.

■ **Note** There is no wildcard that allows all fields to be selected. If you want to receive all the fields for a data type, then you must include all of them in the query.

Click the Execute Query button to send the request to the GraphQL server, which will return the following result:

```
...
{
  "data": {
    "suppliers": [
      {
        "id": "1",
        "name": "Surf Dudes",
        "city": "San Jose",
        "products": ["1","2"]
      },
      {
        "id": "2",
        "name": "Goal Oriented",
        "city": "Seattle",
        "products": ["3","4","5"]
      },
      {
        "id": "3",
        "name": "Bored Games",
        "city": "New York",
        "products": ["6","7","8","9"]
      }
    ]
  }
}
...
```

This may not seem like a huge departure from a REST web service, but even with this basic query, the client is able to select the fields it requires and the order in which they will be expressed.

Querying for Related Data

The GraphQL service is working, and it can be used to get `product` and `supplier` data, which meets the basic needs of the data tables in the example application. However, one of the most powerful features of GraphQL is the ease with which it supports related data in queries, allowing a single query to return results that contain multiple types. In Listing 24-8, I have changed the `products` field to the schema for the `supplier` data type.

GETTING SCHEMA DETAILS FOR GRAPHQL SERVICES

Writing the schema gives the best insight into the queries that a GraphQL service supports, but that isn't always possible. If you are not writing your own schema, the first thing to do is look for developer documentation; many public GraphQL services publish comprehensive schema documentation, such as the GitHub API, described at https://developer.github.com/v4.

Many services also support GraphiQL or similar tools, most of which support schema navigation. GraphiQL, for example, makes it easy to explore the schema through its Docs link, which lets you navigate through the queries and mutations that a service supports.

If there is no documentation and no support for GraphiQL, you can the GraphQL introspection features to send queries about the schema. For example, the following schema query will list the regular queries that a service supports:

```
...
{
     __schema {
    queryType {
      fields {
        name
      }
    }
  }
}
...
```

The special __schema query data type is used to request information about the schema. You can find more details of the GraphQL introspection features at https://graphql.org/learn/introspection.

Listing 24-8. Changing a Data Field in the schema.graphql File in the src/graphql Folder

```
type product {
    id: ID!,
    name: String!,
    category: String!
    price: Float!
}

type supplier {
    id: ID!,
    name: String!,
    city: String!,
    products: [product]
}

type Query {
    products: [product],
    suppliers: [supplier]
}
```

Instead of returning an array of ID values, the products field now returns an array of `supplier` objects. To support this change, I need to process the data used by the resolvers to resolve the relationship between each `supplier` and its related `product` objects, as shown in Listing 24-9.

Listing 24-9. Resolving Related Data in the resolvers.js File in the src/graphql Folder

```
var data = require("../../restData")();
module.exports = {

    products: () => data.products,

    suppliers: () => data.suppliers.map(s => ({
        ...s, products: () => s.products.map(id =>
            data.products.find(p => p.id === Number(id)))
    }))
}
```

The data is processed so that each `supplier` object has a `products` property. The products property is a function that will resolve the related data and that will be invoked only if the client has requested this data field, which ensures that the server doesn't do work to get data that has not been asked for.

Stop the GraphQL server using Control+C and run the command shown in Listing 24-10 in the `productapp` folder to start it again.

Listing 24-10. Starting the GraphQL Server

```
node graphqlServer.js
```

Navigate to `http://localhost:3600/graphql` and enter the query shown in Listing 24-11 into the left pane of the GraphiQL window. This query takes advantage of the change to the GraphQL schema to request suppliers and their related product data in a single query.

Listing 24-11. Querying for Related Data

```
query {
  suppliers {
    id,
    name,
    city,
    products {
      name
    }
  }
}
```

When a field returns a complex type, such as `product`, the query must select the fields that are required. The addition to the query in Listing 24-11 asks the server to provide the `id`, `name`, and `city` fields of each `supplier` object and the `name` field from each of its related `product` objects. Click the Execute Query button, and you will receive the following results:

```
...
{
  "data": {
    "suppliers": [
      {
        "id": "1", "name": "Surf Dudes", "city": "San Jose",
        "products": [{ "name": "Kayak" }, { "name": "Lifejacket" }]
      },
      {
        "id": "2", "name": "Goal Oriented", "city": "Seattle",
        "products": [{ "name": "Soccer Ball" },{ "name": "Corner Flags" },
          { "name": "Stadium" }]
      },
      {
        "id": "3", "name": "Bored Games", "city": "New York",
        "products": [{ "name": "Thinking Cap" },{ "name": "Unsteady Chair" },
          { "name": "Human Chess Board" }, { "name": "Bling Bling King" }]
      }
    ]
  }
}
...
```

Notice that the client specifies the fields required for both the `supplier` objects and the related `product` data, which ensures that only the data required by the application is retrieved.

■ **Note** In addition to regular queries, the GraphQL specification includes support for *subscriptions*, which provide ongoing updates for data that is changing on the server. Subscriptions are not widely or consistently supported, and I don't describe them in this book.

Creating Queries with Arguments

The queries that are currently offered by the GraphQL server allow the user to select the fields that are required but not select the objects in the result, which is a requirement for the requests for individual objects. To give the client the ability to customize requests, GraphQL supports arguments, as shown in Listing 24-12.

Listing 24-12. Using Arguments in the schema.graphql File in the src/graphql Folder

```
type product {
    id: ID!,
    name: String!,
    category: String!
    price: Float!
}

type supplier {
    id: ID!,
    name: String!,
    city: String!,
    products: [product]
}

type Query {
    products: [product],
    product(id: ID!): product,
    suppliers: [supplier]
    supplier(id: ID!): supplier
}
```

Arguments are defined in parentheses after the query name, and each argument is assigned a name and a type. In Listing 24-12, I added queries called `product` and `supplier`, each of which defines an `id` argument whose type is `ID` and which has been denoted as mandatory with an exclamation mark. In Listing 24-13, I have added resolvers for the queries that use the `id` value to select a data object.

Listing 24-13. Defining Resolvers in the resolvers.js File in the src/graphql Folder

```
var data = require("../../restData")();

module.exports = {

    products: () => data.products,

    product: (args) => data.products.find(p => p.id === parseInt(args.id)),

    suppliers: () => data.suppliers.map(s => ({
        ...s, products: () => s.products.map(id =>
            data.products.find(p => p.id === Number(id)))
    })),

    supplier: (args) => {
        const result = data.suppliers.find(s => s.id === parseInt(args.id));
        if (result) {
            return {
```

```
                ...result,
                products: () => result.products.map(id =>
                    data.products.find(p => p.id === Number(id)))
            }
        }
    }
}
```

The resolver function receives an object whose properties correspond to the query arguments. To get the `id` value specified in the query, the resolver functions read the `args.id` property. I can simplify this code by destructing the argument object, as shown in Listing 24-14.

■ **Tip** Notice I used the `parseInt` function to convert the `id` argument for comparison. A direct comparison using `===` between an ID value and a JavaScript Number value will return `false`.

Listing 24-14. Destructing Arguments in the resolvers.js File in the src/graphql Folder

```
var data = require("../../restData")();

module.exports = {

    products: () => data.products,

    product: ({id}) => data.products.find(p => p.id === parseInt(id)),

    suppliers: () => data.suppliers.map(s => ({
        ...s, products: () => s.products.map(id =>
            data.products.find(p => p.id === Number(id)))
    })),

    supplier: ({id}) => {
        const result = data.suppliers.find(s => s.id === parseInt(id));
        if (result) {
            return {
                ...result,
                products: () => result.products.map(id =>
                    data.products.find(p => p.id === Number(id)))
            }
        }
    }
}
```

Restart the GraphQL server and enter the query shown in Listing 24-15 into the GraphiQL window.

Listing 24-15. Querying with an Argument

```
query {
  supplier(id: 1) {
    id,
    name,
    city,
    products {
      name
    }
  }
}
```

This query requests the supplier object whose `id` value is 1 and asks for the `id`, `name`, and `city` fields, along with the `name` field of the related products, producing the following result:

```
...
{
  "data": {
    "supplier": {
      "id": "1",
      "name": "Surf Dudes",
      "city": "San Jose",
      "products": [{ "name": "Kayak" },{ "name": "Lifejacket" }]
    }
  }
}
...
```

Adding Arguments to Fields

Arguments can be defined for individual fields, which allows the client to be more specific about the data it requires. In Listing 24-16, I have added an argument to the schema definition for the `supplier` type, which will allow the client to filter the related product objects by name.

Listing 24-16. Adding a Field Argument in the schema.graphql File in the src/graphql Folder

```
type product {
    id: ID!,
    name: String!,
    category: String!
    price: Float!
}

type supplier {
    id: ID!,
    name: String!,
    city: String!,
    products(nameFilter: String = ""): [product]
}
```

```
type Query {
    products: [product],
    product(id: ID!): product,
    suppliers: [supplier]
    supplier(id: ID!): supplier
}
```

The products field has been redefined to receive a string nameFilter argument. No exclamation point has been used, which means that the argument is optional. If no value is used, the default value of an empty string will be used instead. The implementation of the argument is shown in Listing 24-17.

Listing 24-17. Implementing a Field Argument in the resolvers.js File in the src/graphql Folder

```
var data = require("../../restData")();

const mapIdsToProducts = (supplier, nameFilter) =>
    supplier.products.map(id => data.products.find(p => p.id === Number(id)))
        .filter(p => p.name.toLowerCase().includes(nameFilter.toLowerCase()));

module.exports = {

    products: () => data.products,

    product: ({id}) => data.products
        .find(p => p.id === parseInt(id)),

    suppliers: () => data.suppliers.map(s => ({
        ...s, products: ({nameFilter}) => mapIdsToProducts(s, nameFilter)
    })),

    supplier: ({id}) => {
        const result = data.suppliers.find(s => s.id === parseInt(id));
        if (result) {
            return {
                ...result,
                products: ({ nameFilter }) => mapIdsToProducts(result, nameFilter)
            }
        }
    }
}
```

To support the field argument, the function that resolves the products property on the supplier objects accepts a parameter, which is deconstructed to get the nameFilter value and used to filter the related product objects by name. Restart the GraphQL server and enter the query shown in Listing 24-18 into GraphiQL to see how a field argument is used in a query.

Listing 24-18. Querying with a Field Argument

```
query {
  supplier(id: 1) {
    id,
    name,
    city,
    products(nameFilter: "ak") {
      name
    }
  }
}
```

Click the Execute Query button, and you will see the following results, which show that the related product objects have been filtered so that only those whose name field contains ak are included.

```
...
{
  "data": {
    "supplier": {
      "id": "1",
      "name": "Surf Dudes",
      "city": "San Jose",
      "products": [{ "name": "Kayak" }]
    }
  }
}
...
```

■ **Caution** The methods used to receive field arguments are invoked for every request, which can create a substantial amount of work for the server. Consider using a memoization package for complex results, such as `fast-memoize` (https://github.com/caiogondim/fast-memoize.js).

Because the field argument is applied to the type and not a specific query, the filter can be used in any query for supplier data that includes related product data. Enter the query shown in Listing 24-19 into GraphiQL for a demonstration.

Listing 24-19. Using a Field Argument in Another Query

```
query {
  suppliers {
    id,
    name,
    city,
    products(nameFilter: "g") {
      name
    }
  }
}
```

Click the Execute Query button, and you will see that the related product data for each supplier object in the results has been filtered.

```
...
{
  "data": {
    "suppliers": [
      {
        "id": "1",
        "name": "Surf Dudes",
        "city": "San Jose",
        "products": []
      },
      {
        "id": "2",
        "name": "Goal Oriented",
        "city": "Seattle",
        "products": [{ "name": "Corner Flags" }]
      },
      {
        "id": "3",
        "name": "Bored Games",
        "city": "New York",
        "products": [{ "name": "Thinking Cap" }, { "name": "Bling Bling King"}]
      }
    ]
  }
}
...
```

Making GraphQL Mutations

Mutations are used to ask the GraphQL server to make changes to its data. Mutations are added to the schema using the special `Mutation` type, and there are two broad approaches available, as shown in Listing 24-20.

Listing 24-20. Defining Mutations in the schema.graphql File in the src/graphql Folder

```
type product {
    id: ID!,
    name: String!,
    category: String!
    price: Float!
}

type supplier {
    id: ID!,
    name: String!,
    city: String!,
    products(nameFilter: String = ""): [product]
}
```

```
type Query {
    products: [product],
    product(id: ID!): product,
    suppliers: [supplier]
    supplier(id: ID!): supplier
}

input productInput {
    id: ID, name: String!, category: String!, price: Int!
}

type Mutation {
    storeProduct(product: productInput): product
    storeSupplier(id: ID, name: String!, city: String!, products: [Int]): supplier
}
```

The first mutation, called `storeProduct`, uses a dedicated *input type*, which allows the client to provide values to describe the changes that are required. Input types are defined using the `input` keyword and support the same features as regular types. In the listing, I have defined an input type called `productInput` that has an optional `id` field and mandatory `name`, `category`, and `price` fields. This is broadly duplicative of the `product` type already defined in the schema, which is a common approach because you can't use regular types as the arguments to mutations.

The `storeSupplier` mutation takes a simple approach, which is to define multiple arguments that allow the client to express the details of a data object without requiring an input type. This is an effective approach for basic mutations, but it can become unwieldy for complex mutations. Both mutations produce a result, which provides the client with an authoritative view of the object that has been created or updated as a result of the mutation, expressed using a regular query type. In Listing 24-21, I have added resolvers for the mutations.

Listing 24-21. Implementing Mutations in the resolvers.js File in the src Folder

```
var data = require("../../restData")();

const mapIdsToProducts = (supplier, nameFilter) =>
    supplier.products.map(id => data.products.find(p => p.id === Number(id)))
        .filter(p => p.name.toLowerCase().includes(nameFilter.toLowerCase()));

let nextId = 100;

module.exports = {

    products: () => data.products,

    product: ({id}) => data.products
        .find(p => p.id === parseInt(id)),

    suppliers: () => data.suppliers.map(s => ({
        ...s, products: ({nameFilter}) => mapIdsToProducts(s, nameFilter)
    })),

    supplier: ({id}) => {
        const result = data.suppliers.find(s => s.id === parseInt(id));
```

```
        if (result) {
            return {
                ...result,
                products: ({ nameFilter }) => mapIdsToProducts(result, nameFilter)
            }
        }
    },

    storeProduct({product}) {
        if (product.id == null) {
            product.id = nextId++;
            data.products.push(product);
        } else {
            product = { ...product, id: Number(product.id)};
            data.products = data.products
                .map(p => p.id === product.id ? product : p);
        }
        return product;
    },

    storeSupplier(args) {
        const supp = { ...args, id: Number(args.id)};
        if (args.id == null) {
            supp.id = nextId++;
            data.suppliers.push(supp)
        } else {
            data.suppliers = data.suppliers.map(s => s.id === supp.id ? supp: s);
        }
        let result = data.suppliers.find(s => s.id === supp.id);
        if (result) {
            return {
                ...result,
                products: ({ nameFilter }) => mapIdsToProducts(result, nameFilter)
            }
        }
    }
}
```

The mutations are implemented as functions that receive arguments, just like queries. These mutations use the ID field to determine whether the client is updating an existing object or storing a new one, and they update the presentation data used by the queries to reflect changes. To update a product with the `storeProduct` mutation, restart the server and enter the GraphQL shown in Listing 24-22 into GraphiQL.

Listing 24-22. Using the storeProduct Mutation

```
mutation {
  storeProduct(product: {
    id: 1,
    name: "Green Kayak",
    category: "Watersports",
    price: 290
  }) {
```

```
    id, name, category, price
  }
}
```

Mutations are performed using the `mutation` keyword, which is the counterpart to the `query` keyword used in the previous example. The name of the mutation is specified, along with a `product` argument that provides `id`, `name`, `category`, and `price`. The fields required from the result are then specified, and, in this case, all of the fields defined by a product are selected.

Click the Execute Query button, and you will see the following results:

```
...
{
  "data": {
    "storeProduct": {
      "id": "1",
      "name": "Green Kayak",
      "category": "Watersports",
      "price": 290
    }
  }
}
...
```

To confirm that the mutation has taken effect, execute the query in Listing 24-23 using GraphiQL.

Listing 24-23. Querying Product Data

```
query {
  product(id: 1) {
    id, name, category, price
  }
}
```

When you execute the query, you will see the following results, reflecting the changes made by the mutation:

```
...
{
  "data": {
    "product": {
      "id": "1",
      "name": "Green Kayak",
      "category": "Watersports",
      "price": 290
    }
  }
}
...
```

The process for using a mutation that doesn't rely on an input type is similar, as shown in Listing 24-24.

Listing 24-24. Using a Mutation Without an Input Type

```
mutation {
  storeSupplier(
    name: "AcmeCo",
    city: "Chicago",
    products: [1, 3]
  ){ id, name, city, products {
      name
    }
  }
}
```

When the query is executed, a new supplier will be created, and the following results will be displayed:

```
...
{
  "data": {
    "storeSupplier": {
      "id": "100",
      "name": "AcmeCo",
      "city": "Chicago",
      "products": [{ "name": "Green Kayak" }, { "name": "Soccer Ball" }]
    }
  }
}
...
```

Notice that the mutation uses `id` values in the `product` field to express the relationship between supplier and product objects, but the result includes the product names. The mutation gets its result from the updated presentation data, showing that the result of a mutation need not be directly related to the data it receives.

Other GraphQL Features

To complete this chapter, I am going to describe some useful features that build on those described earlier. These are all optional, but they can be used to make the GraphQL service easier to work with.

Using Request Variables

GraphQL variables are intended to allow a request to be defined once and then customized with arguments each time it is used, without forcing the client to dynamically generate and serialize the complete request data for every operation. The query shown in Listing 24-25 defines a variable that is used as the argument for the product query.

Listing 24-25. A Query with a Variable

```
query ($id: ID!) {
  product(id: $id) {
    id, name, category, price
  }
}
```

Variables are applied to the query or mutation and are defined using a name that starts with a dollar sign (the $ character) and assigned a type. In this case, the query defines a variable called `id` whose type is a mandatory `ID`. Inside the query, the variable is used as `$id` and is passed to the `product` query argument.

To use the variable, enter the query into GraphiQL; expand the Query Variables section, which is at the bottom-left side of the window; and enter the code shown in Listing 24-26.

Listing 24-26. Defining a Value for a Variable

```
{
  "id": 2
}
```

This provides a value of 2 for the `id` variable. Click the Execute Query button, and the query and the variable will be sent to the GraphQL server, with the effect that the product object whose `id` is 2 is selected, as shown in Figure 24-3.

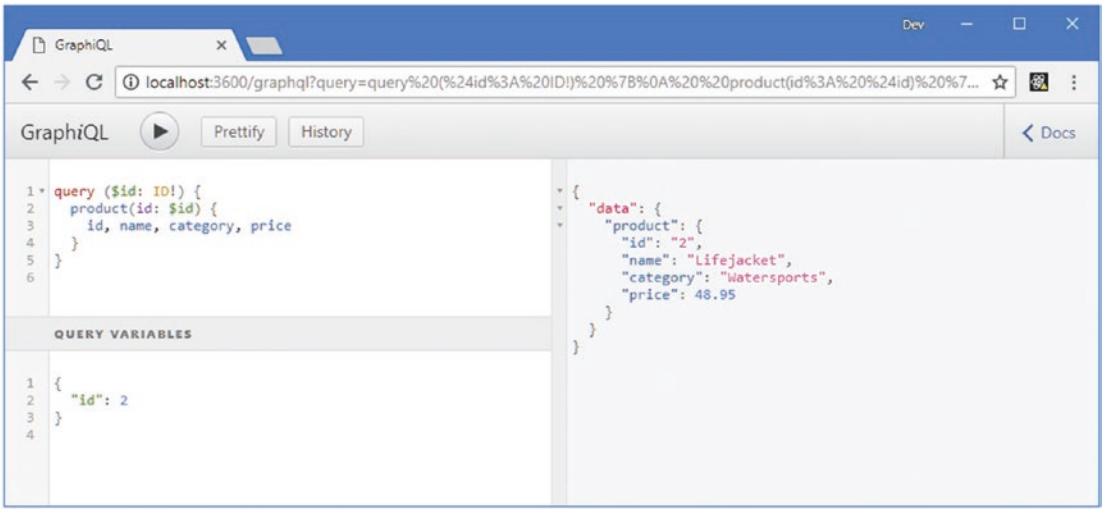

Figure 24-3. *Using a query variable*

Variables may not appear useful when using GraphiQL, but they can simplify client development, as demonstrated in Chapter 24.

Making Multiple Requests

A single operation can contain multiple requests or mutations. Enter the queries shown in Listing 24-27 into the GraphiQL window.

Listing 24-27. Making Multiple Queries

```
query {
  product(id: 1) {
    id, name, category, price
  },
  supplier(id: 1) {
    id, name, city
  }
}
```

Queries are separated by commas and are contained within the outer set of braces, following the `query` keyword. Click the Execute Query button, and you will see the following output, which combines the results of both queries into a single response:

```
...
{
  "data": {
    "product": {
      "id": "1",
      "name": "Kayak",
      "category": "Watersports",
      "price": 275
    },
    "supplier": {
      "id": "1",
      "name": "Surf Dudes",
      "city": "San Jose"
    }
  }
}
...
```

Notice that the name of each query is used to denote its section of the response, making it easy to differentiate between the result from the `product` and `supplier` queries. This can present a problem when you want to use the same query multiple times and so GraphQL supports aliases, which assign a name that is applied to the results. Enter the queries, shown in Listing 24-28, into GraphiQL.

Listing 24-28. Using a Query Alias

```
query {
  first: product(id: 1) {
    id, name, category, price
  },
  second: product(id: 2) {
    id, name, category, price
  }
}
```

The alias comes before the query and is followed by a colon (the : character). In the listing, there are two product queries that have been given the aliases first and second. Click the Execute Query button, and you will see how these names are used in the query results.

```
...
{
  "data": {
    "first": {
      "id": "1",
      "name": "Kayak",
      "category": "Watersports",
      "price": 275
    },
    "second": {
      "id": "2",
      "name": "Lifejacket",
      "category": "Watersports",
      "price": 48.95
    }
  }
}

...
```

Using Query Fragments for Field Selection

The requirement to select result fields from every query can lead to duplication in the client, such as in Listing 24-28, where both the first and second queries select the id, name, category, and price fields. Selections of fields can be defined once using the GraphQL fragments feature and then applied to multiple requests. In Listing 24-29, I have defined a fragment and used it in the queries.

Listing 24-29. Using a Query Fragment

```
fragment coreFields on product {
  id, name, category
}

query {
  first: product(id: 1) {
    ...coreFields,
    price
  },
  second: product(id: 2) {
    ...coreFields
  }
}
```

Fragments are defined using the `fragment` and `on` keywords and are specific to a single type. In Listing 24-29, the fragment is assigned the name `coreFields` and is defined for `product` objects. The spread operator is used to apply a fragment, which can be mixed with regular fields selections. Click the Execute Query button, and you will see the following results:

```
...
{
  "data": {
    "first": {
      "id": "1",
      "name": "Kayak",
      "category": "Watersports",
      "price": 275
    },
    "second": {
      "id": "2",
      "name": "Lifejacket",
      "category": "Watersports"
    }
  }
}
...
```

Summary

In this chapter, I introduced GraphQL. I explained the role of the schema and its resolvers, and I demonstrated the process for creating a simple GraphQL service for static data. I showed you how to define queries to get data from a GraphQL service and how to use mutations to make changes. All of the example in this chapter were performed using the GraphiQL tool, and in the next chapter, I show you how to consume GraphQL in a React application.