GraphQL and Android

Mark L. Murphy

GraphQL and Android
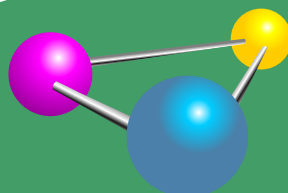
by Mark L. Murphy

CommonsWare

GraphQL and Android
by Mark L. Murphy
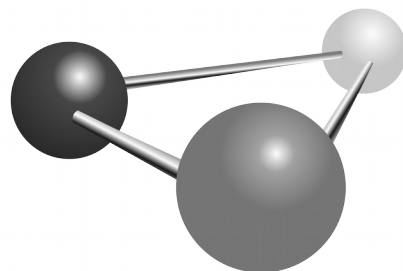
Table of Contents

Headings formatted in ***bold-italic*** have changed since the last version.

Preface

Thanks!

Thanks for your interest in Android app development, the world's most popular operating system! And, thanks for your interest in GraphQL, an increasingly-popular option for communications between clients and servers.

And, most of all, thanks for your interest in this book! I sincerely hope you find it useful!

But, however, bear in mind that it will not be funny. Not one little bit. This book will be completely serious, without any jokes or other forms of humor. Honest.

(well, OK, perhaps not)

First-Generation Book

Android app development can be divided into two generations:

- First-generation app development uses Java as the programming language and leverages the Android Support Library and the `android.arch` edition of the Architecture Components
- Second-generation app development more often uses Kotlin as the programming language and leverages AndroidX and the rest of Jetpack (which includes an AndroidX edition of the Architecture Components)

This book is a first-generation book. It explores the `android.arch` edition of the Architecture Components and it uses Java for most of the examples.

The material in this book is largely unchanged from August 2017.

Prerequisites

This book is targeted at intermediate Android app developers — those with some experience but not necessarily "experts" in the field.

How the Book Is Structured

We start off with a quick spin through making GraphQL requests, using canned tools and custom Android apps. These chapters are designed to help you explore the basics of exchanging data via GraphQL, plus learn about how GraphQL compares with other ways that you have used to exchange data (e.g., REST-style Web services).

Next, we take a deep dive into GraphQL syntax. A GraphQL server publishes a schema, indicating what we can request or modify. In this series of chapters, we will explore various facets of that schema and learn how we can request or modify that data given the restrictions imposed by the schema.

Overall, the hope is that this book will give you thorough grounding in how to integrate your app with GraphQL servers.

Source Code and Its License

The source code samples shown in this book are available for download from the [book's GitHub repository](). All of the Android projects are licensed under the [Apache 2.0 License](), in case you have the desire to reuse any of it.

Copying source code directly from the book, in the PDF editions, works best with Adobe Reader, though it may also work with other PDF viewers. Some PDF viewers, for reasons that remain unclear, foul up copying the source code to the clipboard when it is selected.

Creative Commons and the Four-to-Free (42F) Guarantee

Each CommonsWare book edition will be available for use under the [Creative Commons Attribution-Noncommercial-ShareAlike 4.0]() license as of the fourth

anniversary of its publication date, or when 4,000 copies of the edition have been sold, whichever comes first. That means that, once four years have elapsed (perhaps sooner!), you can use this prose for non-commercial purposes. That is our Four-to-Free Guarantee to our readers and the broader community. For the purposes of this guarantee, new Warescriptions and renewals will be counted as sales of this edition, starting from the time the edition is published.

This edition of this book will be available under the aforementioned Creative Commons license on *1 January 2023*. Of course, watch the CommonsWare Web site, as this edition might be relicensed sooner based on sales.

For more details on the Creative Commons Attribution-Noncommercial-ShareAlike 3.0 license, visit [the Creative Commons Web site](#)

Note that future editions of this book will become free on later dates, each four years from the publication of that edition or based on sales of that specific edition. Releasing one edition under the Creative Commons license does not automatically release *all* editions under that license.

Acknowledgments

The author would like to thank [Lee Byron](#) and the rest of the team that developed GraphQL and shepherds its ongoing development.

The author would also like to thank [Mike Nakhimovich](#) and the rest of the team that developed Apollo-Android, a key open source library for using GraphQL on Android that is profiled in this book.

Introducing GraphQL

GraphQL in Five Minutes

Before we get into what GraphQL is, why you might be using it, and what it has to do with Android, let's spend a few minutes playing around with GraphQL. One of the nice things about this piece of technology is that you can experiment with it easily enough, using publicly-available GraphQL sites... starting from the comfort of your desktop Web browser.

(you could use a mobile Web browser, such as on your Android phone or tablet, but a physical keyboard is *really* handy when working on programming-like tasks)

Step #1: Gotta Get a GraphiQL

Developers love tools. After all, programming computers by hand-calculating machine instructions for a CPU gets to be tiresome rather quickly. We are used to having IDEs for base development, with structured editors, debuggers, and the like. We are used to having databases with clients that we can use for testing queries. We are used to being able to browse the source of Web pages and bring up other interactive Web development tools in the browser.

The people who created GraphQL provided a Web-based interactive GraphQL tool called GraphiQL. GraphiQL allows you to play around with GraphQL without having to install anything or even know much about GraphQL.

With that in mind, in a desktop Web browser, visit https://graphql-demo.commonsware.com/0.3/graphql. While this link will work on a phone or tablet, GraphiQL is not designed for use from mobile devices.

This will bring up GraphiQL on a CommonsWare-hosted GraphQL-powered server:
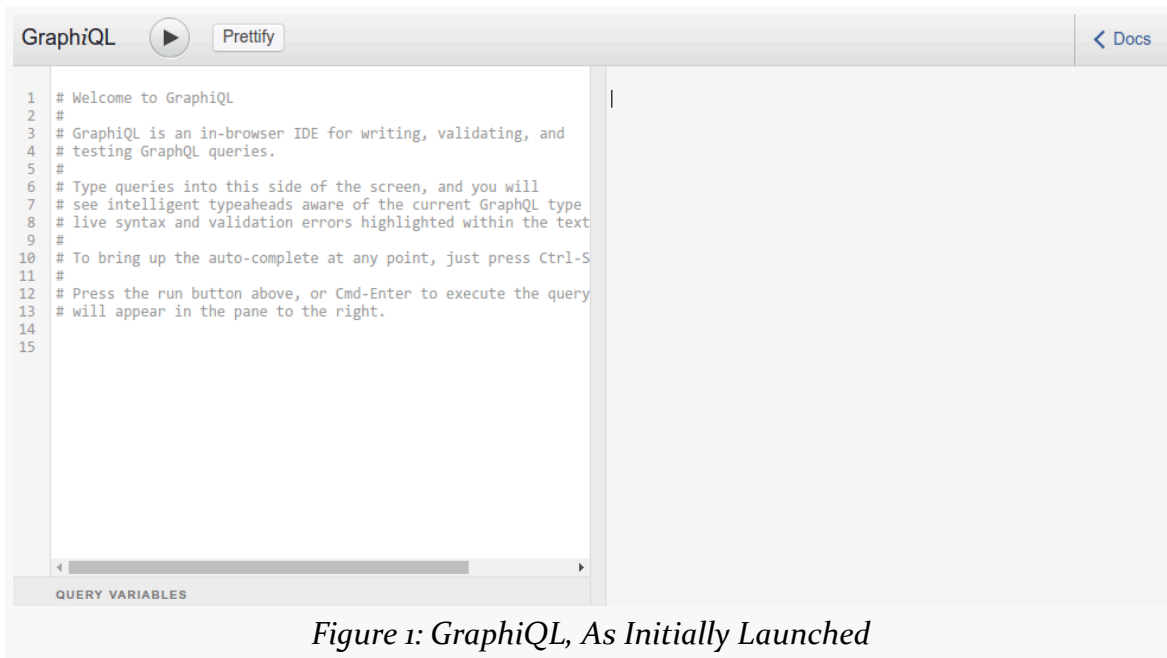
Figure 1: GraphiQL, As Initially Launched

On the left, you can enter in something in GraphQL. Clicking the "run" button (rightward-pointing triangle) sends that to the server, which triggers a response on the right.

Step #2: Drafting a Document

So, on the left, type in:

```
{
  allTrips {
    id
    title
  }
}
```

You will notice that as you start typing, GraphiQL will offer a code-completion popup:
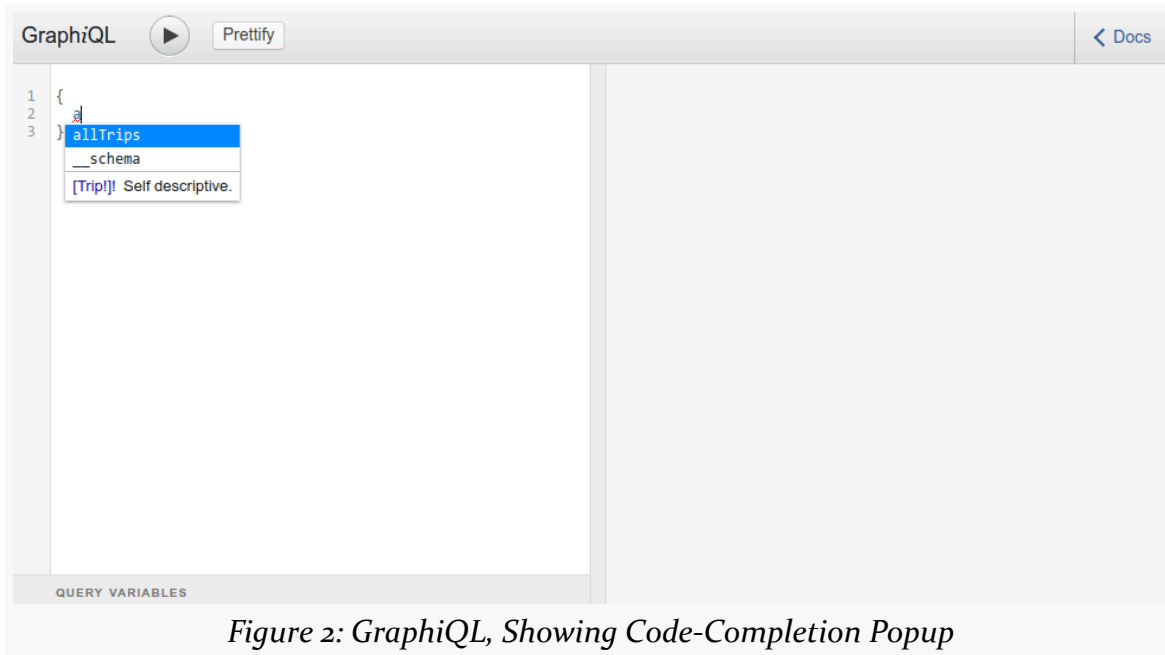
Figure 2: GraphiQL, Showing Code-Completion Popup

Similarly, if you get part-way through — just entering in the `allTrips` bit without its own set of braces — you will see that the `allTrips` gets a red under-squiggle, indicating an error that you can view by hovering your mouse over it:
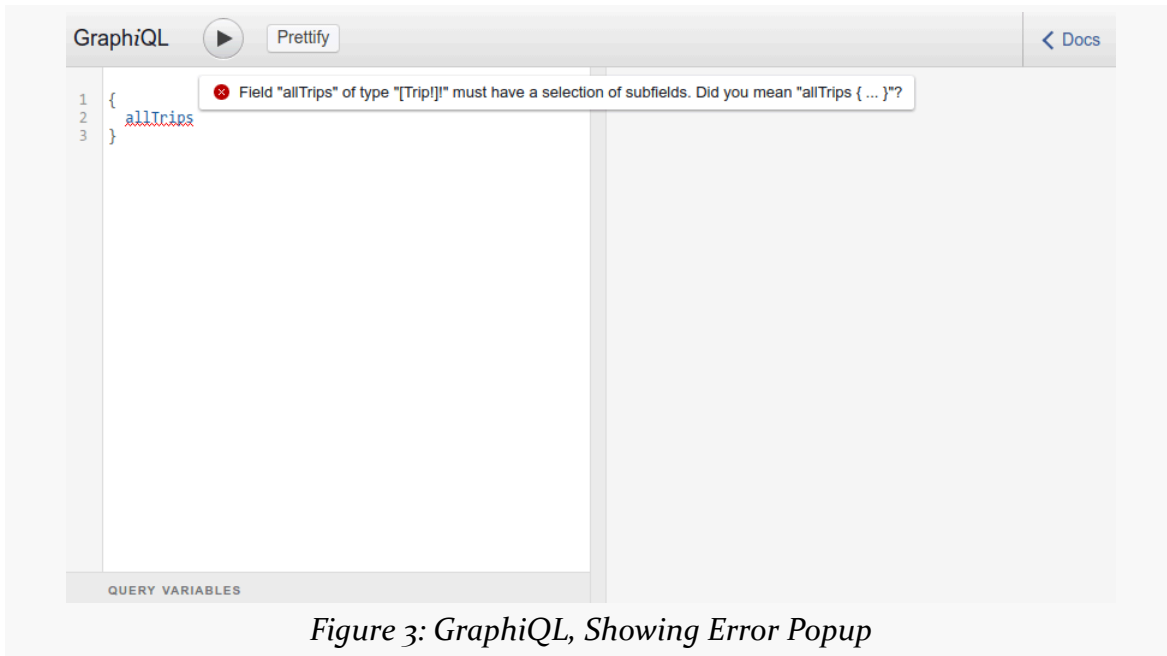
Figure 3: GraphiQL, Showing Error Popup

As a result, this feels a bit like working in an IDE like Android Studio.

This is all possible because GraphiQL knows some stuff about the data being published via GraphQL from the server. We will come back to that point in a bit.

Step #3: Making the Request

After you have typed in the full "document" shown above, click the "Run" button, which is the rightward-pointing triangle in the circle. You should see some JSON appear on the right:

Figure 4: GraphiQL, Showing Results

```
{
  "data": {
    "allTrips": [
      {
        "id": "2c494055-78bc-430c-9ab7-19817f3fc060",
        "title": "Vacation!"
      },
      {
        "id": "e323fed5-6805-4bcf-8cb6-8b7a5014a9d9",
        "title": "Business Trip"
      }
    ]
  }
}
```

(the JSON may be somewhat different than what is shown above, but it should be similar)

Congratulations! You just created and executed a GraphQL request! Cheese and cake for everyone!

(note: you will need to supply your own cheese and cake)

Step #4: Looking at the Docs

Now, if you look at the GraphiQL navigation bar, you will see a "Docs" button in the upper right corner. Like all good developers, you certainly intend to read the documentation.

Right?

Right?!?

So, click that "Docs" button, which will open up a panel on the right:
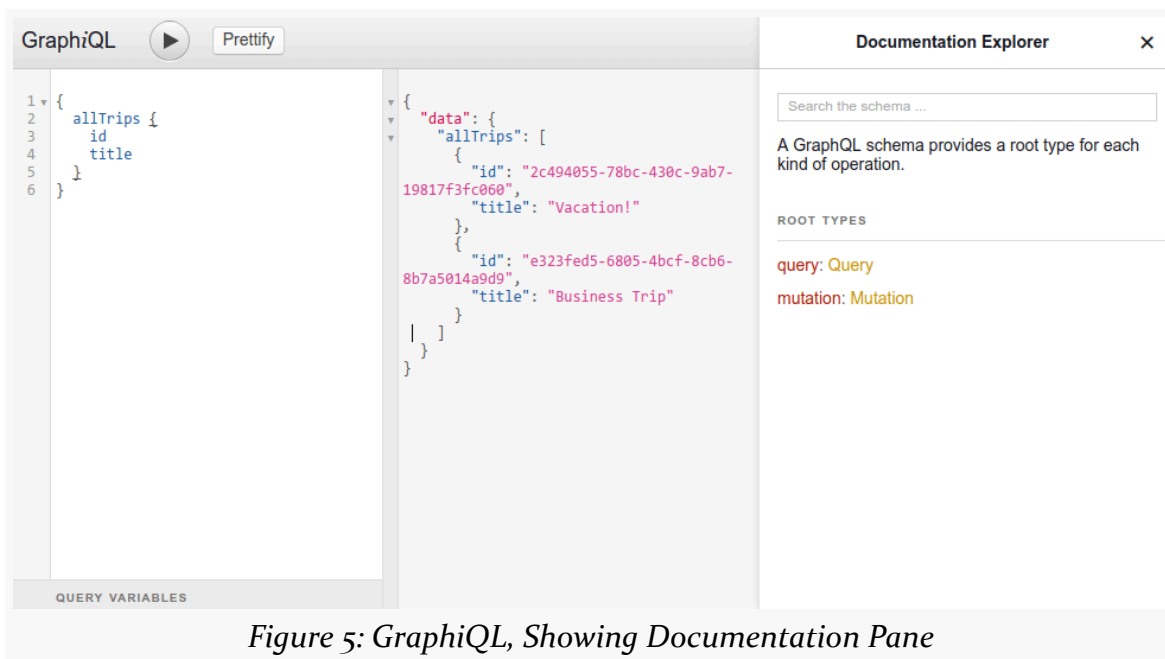
Figure 5: GraphiQL, Showing Documentation Pane

You might expect this to be documentation about the GraphiQL tool, or perhaps documentation about GraphQL. Instead, it is documentation about the data being published by the server.

For example, you can click on the Query link to view all of the things that we can

request from the server:
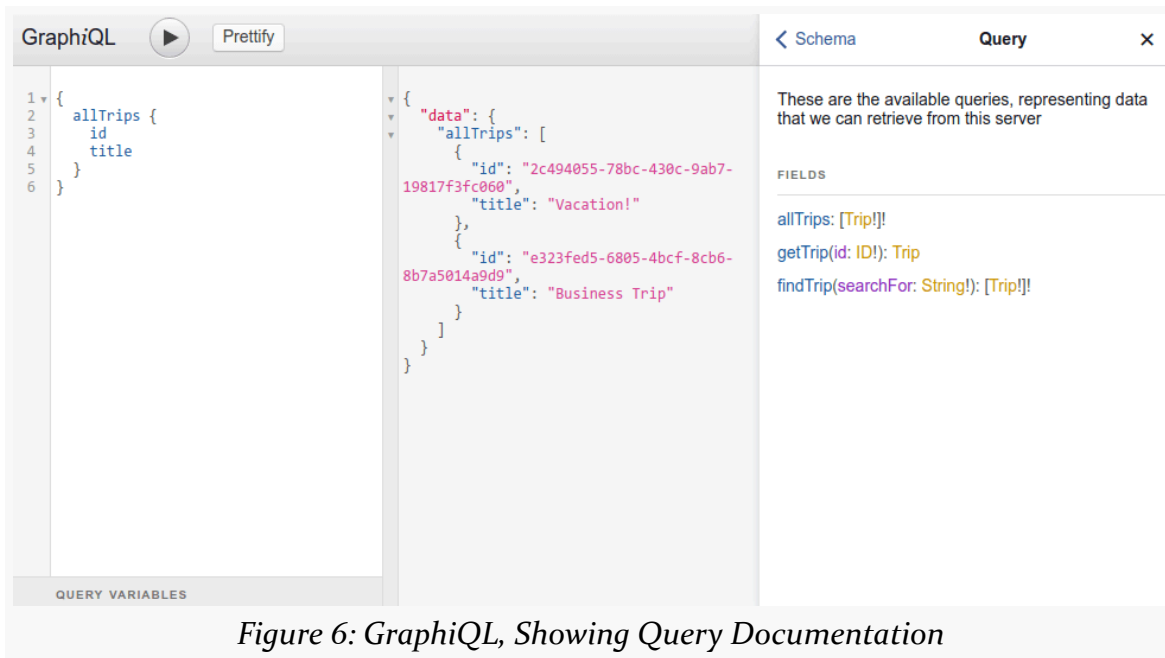
Figure 6: GraphiQL, Showing Query Documentation

Similarly, clicking on the `Trip` link will provide details about what we can retrieve from the results returned by `allTrips`:
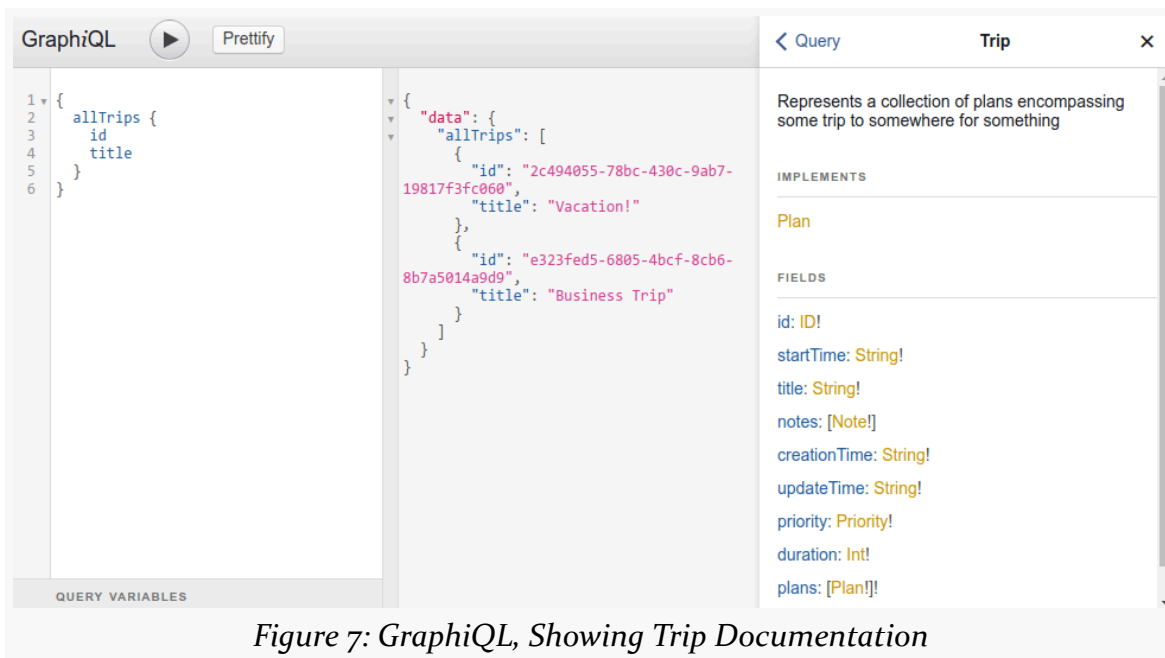
Figure 7: GraphiQL, Showing Trip Documentation

What is interesting about this documentation is that it does not really exist. Nobody wrote this documentation as a set of Web pages. Instead, it is being generated on the fly, by the GraphiQL tool. GraphiQL is asking the server about the sorts of data that it publishes, and that information is being used to create the documentation. This "introspection" capability of GraphQL is also what powers the code-completion popups when you were typing in the GraphQL document earlier. We will explore how you can use introspection, perhaps as part of building some GraphQL tools of your own, [later in the book](#).

But now that you have a taste of GraphQL, let's continue to the next chapter and see what all the fuss is about.

The Role of GraphQL

Why was GraphQL created?

What does GraphQL offer us that alternatives do not?

What are those alternatives, anyway?

GraphQL has garnered a fair bit of attention in its short lifetime, but mostly in the world of Web developers. In the world of mobile app development, though, GraphQL has had a slower adoption curve and a corresponding lower amount of discussion.

In other words: you've got questions. That's understandable.

In this chapter, we will focus on trying to get you answers to those questions, so that you better understand the reasons why you might want to use GraphQL, beyond "my boss told me so".

However, let's first focus on another question…

So, What Did We Just Do?

In [the preceding chapter](#), we played around with a Web-based tool called GraphiQL, typed some stuff in, and got some stuff back. To anyone who has used the Internet in the past decade or so, what we did, on the surface, may not seem that exciting.

And, to some extent, GraphQL is not breaking new ground. It is merely a way to describe a request for some data, and a way for sources of data to respond to those requests and structure a response. We have been doing that sort of thing for

decades, picking up steam with the advent of [client-server](#) as a popular system paradigm in the late 1980's. There is a decent chance that you were not even born yet.

But let's think a bit about the nature of what we typed in and the nature of what we got back.

First, let's look at what we typed in again:

```
{
  allTrips {
    id
    title
  }
}
```

GraphQL is all about the fields. `allTrips` is a field on a particular object, known as the `Query`, that exposes data that we can read from the server. As it turns out, and as we saw in the generated documentation, `allTrips` will return data about `Trip` objects. `Trip` objects have their own fields, like `id`, `title`, and `creationTime`. When we get the `allTrips` response, rather than getting all possible fields on the `Trip` objects, we are saying "only give us the `id` and `title`, please".

Now, let's compare that with the response:

```
{
  "data": {
    "allTrips": [
      {
        "id": "2c494055-78bc-430c-9ab7-19817f3fc060",
        "title": "Vacation!"
      },
      {
        "id": "e323fed5-6805-4bcf-8cb6-8b7a5014a9d9",
        "title": "Business Trip"
      }
    ]
  }
}
```

First, the response is in JSON. That is not a requirement of GraphQL, though it is a common implementation. So, this feels a bit like a REST-style Web service, or perhaps some sort of document database.

However, the specific data we got back was structured to match the data that we requested:

- The JSON attributes that we got back match those that we requested
- We seem to be getting back trips, and we asked for `allTrips`, which we saw in the generated documentation has something to do with a `Trip` object

This feature of limiting the breadth of the response to only the fields that we care about is interesting. This is not the sort of thing that you usually see with REST-style Web services, or even things like Java method calls. For example, in Java, we could have something like:

```java
List<Trip> allTrips() {
  // really fun code here
}
```

However, there, the `Trip` "is what it is". We do not tell `allTrips()` what particular bits of data are of interest.

Instead, this feels a bit more like something that we might do with SQL, to request data from a relational database:

```sql
SELECT id, title FROM trips;
```

GraphQL borrows bits and pieces from a number of existing communication patterns to craft a new one. The key for us is whether this new one is a case of "the whole is greater than the sum of its parts", or whether this new one more resembles the monster created by Dr. Frankenstein (either [Victor Frankenstein](#) or [Frederick Frankenstein](#), as you see fit).

What Exactly is GraphQL?

GraphQL is a query language designed to be sent to some server, where the server can send back a response based upon the request that was submitted.

Of course, that explanation is a bit broad.

A Specification

The most important thing that GraphQL "is" is a specification.

11

[That specification](#) describes:

- What the syntax is for the query language
- How, in general terms, the server should interpret the request
- What the server response should be, in terms of syntax, based upon what was in the request

Along the way, the specification enforces a few design principles, which we will see [shortly](#).

A Set of Conventions

However, the GraphQL specification is fairly compact. It focuses on syntax and expected results. It does *not* address a lot of things that one needs to really apply GraphQL in practice.

The biggest one is: how does the request get to the server, and how does the response get back?

When we executed the request using GraphiQL, we used HTTP for those steps. GraphiQL knows:

- the URL from its server for sending requests
- how to take what we typed in and supply it to the server on that URL
- how to interpret the HTTP response

However, the GraphQL specification does not address this.

On the one hand, this allows for a lot of flexibility and room for experimentation. On the other hand, it does make it a bit difficult for newcomers to understand how to use GraphQL to accomplish something worthwhile.

As with many software ecosystems, conventions take over where specifications leave off. In the case of determining how we send GraphQL and responses between clients and servers, the GraphQL developer documentation describes [conventions for serving GraphQL over HTTP](#).

Other conventions revolve around certain GraphQL-aware products and projects. For example, you will hear about "Relay-compliant servers", referring here to Facebook's [Relay project](#), which ties GraphQL to a particular JavaScript framework for creating Web apps.

A Lot of Froth

Whenever you have a new, exciting technology, all sorts of people start offering things around the technology: tools, libraries, consulting, conferences...

Even books!

These each provide possibilities for furthering your use of the technology. However, until enough people have "kicked the tires" on them and some consensus opinions start forming, it may be difficult to distinguish what might turn into long-term successes and what might only be useful for a short while.

Even books have a limited life, insofar as the technology may advance and make portions of the book obsolete. That is why this book will be updated periodically, to try to keep it up to date with the latest changes.

GraphQL Design Principles

In its overview, the GraphQL specification offers up a handful of design principles that are being used to help guide the development of GraphQL itself. Understanding these can help us make sense of why GraphQL works the way it does.

Hierarchical

Turn back to GraphiQL for a moment and execute this request:

```
{
  allTrips {
    id
    title
    plans {
      id
      title
    }
  }
}
```

Notice how the document specifies a hierarchy of information that we want back. A `Trip` contains "plans", and we are asking for the `id` and the `title` for those plans.

The server response mimics that hierarchy:

```json
{
  "data": {
    "allTrips": [
      {
        "id": "2c494055-78bc-430c-9ab7-19817f3fc060",
        "title": "Vacation!",
        "plans": [
          {
            "id": "319185bd-fab0-49e3-86ce-251d2aaa5d23",
            "title": "Flight to Chicago"
          },
          {
            "id": "319185bd-fab0-49e3-86ce-251d2aaa5d23",
            "title": "House of Munster"
          }
        ]
      },
      {
        "id": "e323fed5-6805-4bcf-8cb6-8b7a5014a9d9",
        "title": "Business Trip",
        "plans": [
          {
            "id": "d40eb2e7-3211-422e-858c-403cbe3fa680",
            "title": "Flight to Denver"
          },
          {
            "id": "e28a591b-cdc9-4328-9e79-9e4ed60ae7d2",
            "title": "Hotel Von"
          }
        ]
      }
    ]
  }
}
```

The "graph" of GraphQL does not refer to bar charts, gridded paper, etc. It refers to an object graph. More accurately, GraphQL allows clients to request a hierarchy of data, which the server sends back.

(perhaps this should be called HierarchyQL, but that is longer to type and more challenging to spell)

This approach stands in contrast to others — like REST — where the server dictates what the responses will look like, with limited client-side configurability.

Client-Focused

Some means of accessing data allows the client the flexibility to state what it wants. SQL, for example, allows the client to stipulate what columns it wants from what tables, what subset of rows to retrieve, how to join the data across those tables, and so on.

Other means of accessing data offer much less flexibility. With many REST-style Web services, for example, you get whatever the server wants to send you, whether or not that strictly meets the clients' needs. You may be able to constrain the response's depth via the URL — a REST-style request for `/v0.3/invoice/34934/lines` might return the line items for an invoice with an ID of 34934. However, you may not be able to ask for two invoices, or only for certain pieces of data about the line items.

Perhaps the key bit in the first sentence of the preceding paragraph is the placement of the apostrophe: "clients'", rather than "client's". Some systems have multiple clients: Web apps, Android apps, iOS apps, other servers, etc. Other systems have but a single client. Many REST-style Web services wind up treating all of these the same, offering a single API, even if that means one client is favored over others in terms of the way the API is structured.

GraphQL is not quite as expressive as SQL. However, it comes far closer to the SQL experience than do many REST-style Web services. As we have seen, with GraphQL, the client indicates what data it wants back, even if that data is but a subset of what the server is in position to offer (e.g., only some attributes, rather than all of them).

Overall, the engineers behind GraphQL want to give servers the freedom to stipulate "the realm of the possible", but for individual clients to be able to request, in a flexible fashion, what portion of "the possible" those clients want.

Strongly Typed

Something is "strongly typed" if data has to be of a declared data type, whether something simple (e.g., string, integer) or something complex (e.g., object-style structure). We often say that programming languages like Java are "strongly typed", in contrast to languages like JavaScript, where any variable can hold any data type. The fact that GraphQL has grown largely out of JavaScript-centric Web app development may be the reason why the GraphQL team emphasizes the fact that GraphQL is strongly typed, as this may run counter to expectations.

The key here is the ability to do development-time validation of GraphQL. It should be possible for development tools to detect invalid GraphQL queries, before they get submitted to the server. Moreover, this should be based not only on whether the queries are syntactically valid (e.g., do we have all the braces in the right spots?), but also whether the types we try using match the types that the server will expect (e.g., trying to provide a string where an integer is expected).

Introspective

Part of what powers this is introspection. GraphQL servers not only can be queried for data but also for *metadata*. Specifically, a client can load information about the "schema" that the server uses, indicating what sorts of things a GraphQL client can ask for. This is reminiscent of some SQL databases, such as the SQLite used in Android, where we can make `.schema` queries to return the structures of tables.

Introspection is powerful for use with tools. In GraphiQL, introspection is what powers:

- Code-completion tooltips
- Document validation
- Auto-generated online documentation

In the libraries and tools that we will see in this book, introspection also can power things like code generation.

Key GraphQL Features

Beyond the official design principles, GraphQL has a number of other positive benefits for developers, mobile app developers in particular.

Platform Neutral

GraphQL clients and servers can run on any platform that you like. There is nothing intrinsic to GraphQL that requires any particular operating system or app development environment. GraphQL *leans* towards Web development, both in terms of data formats (JSON is the common response format) and in terms of available tools and libraries. However, there is nothing stopping you from writing a native app for Android (or iOS, or Windows, etc.) using GraphQL, though the amount of client-side support varies a bit by platform.

That being said, GraphQL is driven a lot by the open source community. As a result, you find more support for platforms where somebody "had an itch to scratch". So, for example, you will find lots more support for writing GraphQL clients and servers in JavaScript than in, say, Visual Basic.

Protocol Neutral

This book, and most of the GraphQL world, tends to think in terms of GraphQL being used with HTTP(S). You transmit the GraphQL request to the server via an HTTP GET or POST operation, and you get a response back based upon that request.

However, the GraphQL specification avoids tying GraphQL to any particular protocol. There is nothing stopping you from using GraphQL over something else (e.g., Web sockets, XMPP) if that fits your particular use case. Once again, a *lean* towards HTTP(S) does not preclude alternatives, though you will be more "on your own" if you use some other protocol.

Storage Neutral

On the server side, GraphQL server libraries generally make no assumptions about where the data is stored. Some frameworks exist to help you serve GraphQL from data in SQL databases. One would imagine that document databases (e.g., CouchBase) and NoSQL databases (e.g., MongoDB) will be popular options. There are even proxies that help you map existing REST-style Web services to GraphQL. And many developers are using GraphQL to wrap around a whole series of microservices, each perhaps with varying APIs, to provide a uniform API for clients to work with.

Similarly, on the client side, you get JSON (or, in principle, some other agreed-upon format), no different than you would from a REST-style Web service. You can cache this as you see fit, whether in relational databases (e.g., SQLite on Android), simple files, memory-only caches, or something else entirely.

Bandwidth-Friendly

GraphQL helps clients and servers save bandwidth in two ways:

1. The client specifies what it needs, so the server does not wind up sending extra bits that the client would wind up throwing out anyway.

17

2. Compared to record-centric options (e.g., SQL, most REST-style Web services), the hierarchical nature of GraphQL may allow you to replace lots of individual calls with one larger one, saving lots of HTTP(S) round trips. For example, retrieving information about an invoice might take two SQL queries or REST calls: one to get the general information (e.g., customer, date, invoice number) and one to get the line items for the invoice (i.e., what the customer actually bought). With GraphQL, that could be retrieved in one request.

GraphQL Compared To…

Making comparisons, for the purposes of trying to explain what something is like, tends to get overdone in many areas ("it's like Uber, but for pet care!"). Still, there can be some merit in comparing and contrasting different data access options with GraphQL.

…REST Web Services

What tends to get compared the most with GraphQL is the REST approach to developing Web services. The comparison seems apt, in that both REST and GraphQL allow clients to retrieve structured data from a server over HTTP(S), frequently in JSON format.

However, REST is heavily tilted towards "CRUD" (create-read-update-delete) operations, tied to associated HTTP verbs. You GET a resource identified by some URL (read). You PUT back to that same URL to update it (update). You DELETE that same URL to get rid of it (delete). You typically POST to some URL to create something related to that resource (create). And that is pretty much it, in terms of possible things to do. And, the HATEOAS ("Hypermedia as the Engine of Application State") aspect of official REST uses URLs and MIME types to tie one resource to another (e.g., an invoice line item back to its invoice).

GraphQL, in contrast:

- Does not use URLs as identifiers
- Does not use HTTP verbs as operations (e.g., PUT for update)
- Does not provide any particular restriction on how to relate different pieces of data (e.g., no HATEOAS equivalent)

Now, in truth, not all REST-style Web services necessarily follow all the REST rules.

Plenty of Web services advertised as being REST do not follow HATEOAS. The rules for what POST does, or how to create new resources, tends to get muddled. In part, this is because REST does not really have a formalized specification. It is more a set of guiding principles, for which a wide range of implementations have been created, each adhering to those principles in varying degrees.

However, there is little question that GraphQL is not REST.

…RPC Web Services

What GraphQL more closely resembles is a Web-based remote procedure call (RPC) system.

Unless you have been doing software development for a couple of decades, it is entirely possible that you have never really dealt with RPC. In effect, RPC is "call functions like you normally do in programming, but across a network". Usually, the RPC framework combines an interface definition language (IDL) with tools that code-generate the client and server sides of the RPC relationship. Clients and servers that use the generated code "feel" like they are working locally, with the framework handling all the issues of transmitting the requests and responses between the client and the server. If you have worked with Android remote services and AIDL, AIDL was designed to make Android inter-process communication (IPC) feel like RPC.

Some RPC systems, like CORBA, have been around so long that they pre-date the Web. Others, like XML-RPC and SOAP, stem from our earliest efforts at creating standardized Web service APIs. More recently, Google released gRPC as another modern take on RPC.

Like RPC frameworks:

- GraphQL defines "functions" that can be invoked, in the form of queries and mutations
- Those "functions" take arguments and return results
- GraphQL libraries handle the low-level "plumbing" of getting requests and responses across the Internet

Perhaps the biggest extension that GraphQL adds over RPC is specifying the data to be returned as a result of the "function call". With RPC, you get whatever the server wants to give you. With GraphQL, the client specifies what portion of the result is of relevance and should be returned. While you could add a similar sort of structure onto RPC, it is a native part of GraphQL.

…SQL Databases

The idea that the client stipulates what structure the server should return sounds a lot like a SQL database. With SQL, the client specifies the desired columns from specific tables. Assuming that the request is well-formed, the server will return exactly the columns that the client asks for, much as how a GraphQL server will return a JSON structure mirroring what was requested.

However, in many respects, that is the only similarity between GraphQL and a SQL database:

- SQL databases support a relatively fixed set of operations (`INSERT`, `UPDATE`, `DELETE`), whereas GraphQL allows the server to publish more customized operations
- SQL databases usually are designed to work over a local network, not over the Internet, the way GraphQL and other Web services are
- SQL databases usually return simple tabular structures (rows of columns), as opposed to the more hierarchical JSON that you can get from a GraphQL server

…Document and NoSQL Databases

Getting a rich JSON structure back from a request sounds a lot like modern document and other types of "NoSQL" databases. Couchbase (and its CouchDB open source counterpart) and MongoDB are examples of document databases, while Cassandra and Redis are examples of other types of NoSQL databases.

Since many of these arose during the rapid growth of Internet services, it is not terribly surprising that those that offer a Web service API support JSON as a data format. However, not all offer a Web service API, delegating that responsibility to Web apps. Beyond that, while GraphQL has a specific structure for requests and responses — a structure that any client or server could support — most of the NoSQL databases (document databases and others) have their own unique request and response formats. There are few standards in this area.

Top-Level GraphQL Terms

Before we get too much farther into GraphQL, we need to establish some definitions of some key terms.

Document

The GraphQL that you have typed into GraphiQL is called a GraphQL document.

In GraphiQL, from your standpoint, that document happens to be the contents of a text editor widget. In traditional programming environments, the document might:

- Be a standalone file that is part of your project
- Be a hard-coded string in your source code
- Be obtained on the fly, which is how the GraphiQL JavaScript code considers what you typed in

Operation

What we typed in was not just a GraphQL document, but a GraphQL operation within that document.

An operation indicates a specific request that we want to make of a GraphQL server or other GraphQL processor. Operations come in two forms:

- Queries, for retrieving data
- Mutuations, for manipulating data and retrieving some data after those manipulations have been performed

Operation Name

We typed in a query:

```
{
  allTrips {
    id
    title
  }
}
```

This is a shorthand notation, resulting in an anonymous query. More often, you will provide two other pieces of information: a keyword (query or mutation) and an operation name:

```
query all {
  allTrips {
    id
    title
  }
}
```

A document can contain one or more operations. If it has only one operation, it can be a query in the anonymous shorthand notation. Otherwise, all operations need the keyword and an operation name.

Roughly speaking, an operation name is equivalent to a function name or method name in other programming languages. It is your local name for this operation. The key here is "local", as the GraphQL server does not really do much with this name.

An operation name is made up of letters (A-Z and a-z), numbers (0-9), and underscores (_). However, it cannot start with a number. In other words, the naming rules for a Java field or method — or an Android resource — apply here as well.

Once we have a document with more than one operation, GraphiQL will ask us to choose which operation to run when we click that "run" button. So, for example, enter in the following GraphQL document into GraphiQL:

```
query all {
  allTrips {
    id
    title
  }
```

```
}

query redundant {
  allTrips {
    id
    title
    priority
  }
}
```

Here, we have a GraphQL document with two `query` operations: `all` and `redundant`. When we click the "run" button, we get a drop-down menu to indicate which of those two operations we wish to run:
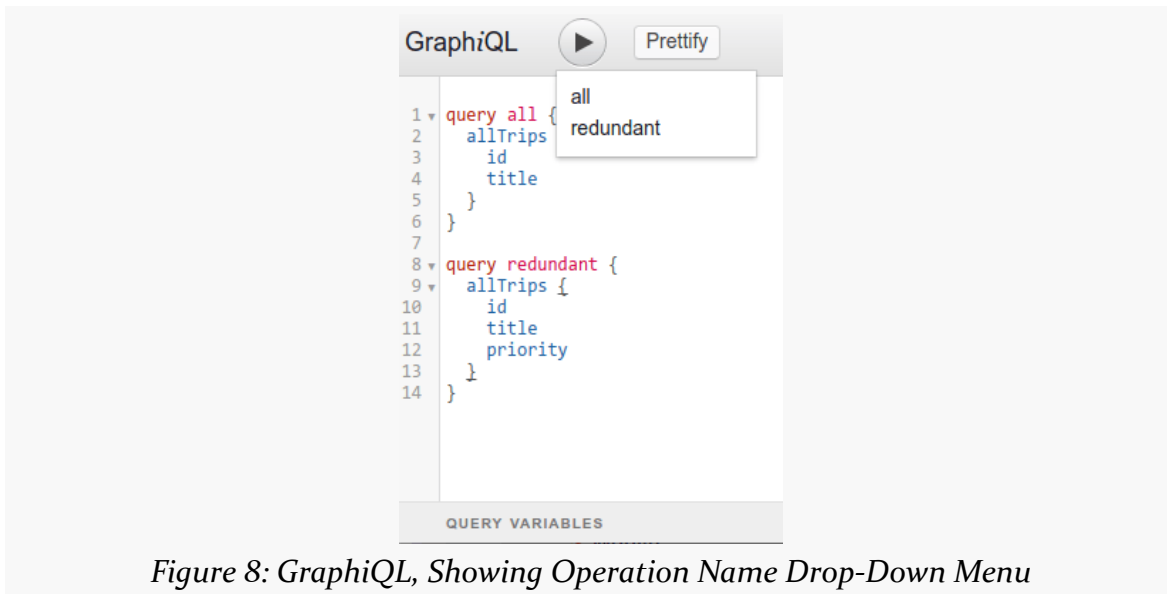
Figure 8: GraphiQL, Showing Operation Name Drop-Down Menu

You have to click on one of those, after which GraphiQL will pass the entire document, plus that operation name, to the server for processing:
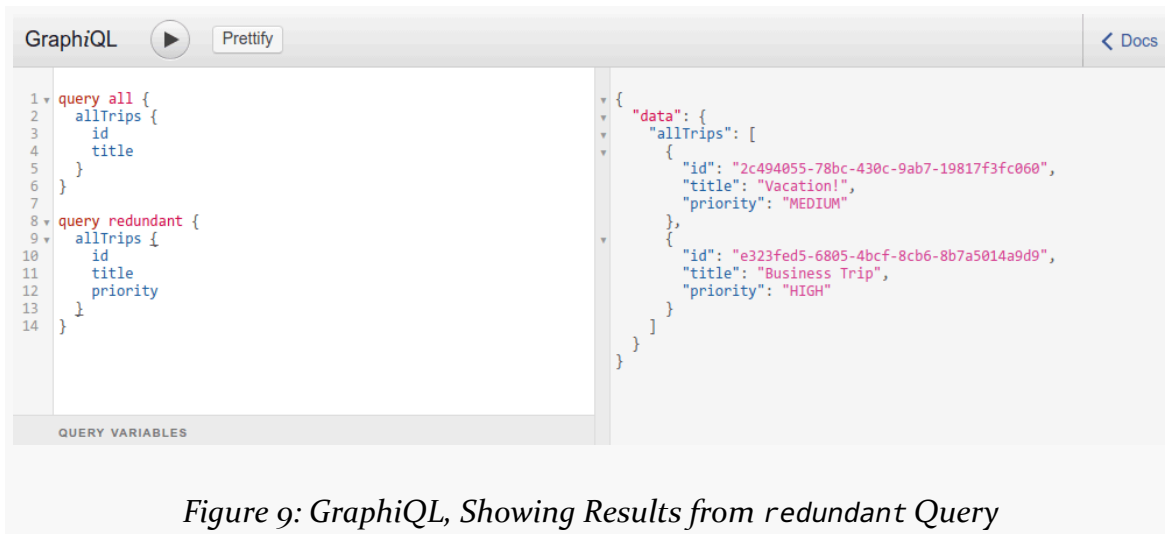
Figure 9: GraphiQL, Showing Results from `redundant` Query

If you have your GraphQL in a dedicated document file, you might elect to put *all* the GraphQL in that file, then use that document and the operation name when working with the server. Or, you could arrange that each GraphQL document contains just one operation, in which case you can skip sending the operation name to the server, as it will know by default which operation to run (since there is only one).

Arguments and Variables

Sometimes, we will be able to pass arguments in as part of an operation:

```
query find {
  findTrips(searchFor: "ca") {
    id title startTime priority duration
  }
}
```

Here, we are asking for `findTrips`, which takes a `searchFor` argument, which we can use for our search criteria (e.g., trips with "ca" in the title). If you run this in the demo server's GraphiQL, you will get a list of matching trips:

```
{
  "data": {
    "findTrips": [
```

```
      {
        "id": "2c494055-78bc-430c-9ab7-19817f3fc060",
        "title": "Vacation!",
        "startTime": "2017-12-20T13:14:00-05:00",
        "priority": "MEDIUM",
        "duration": 10080
      }
    ]
  }
}
```

In addition to hard-coding the argument value into the document, though, we can use variables.

For example, in the demo server's GraphiQL, enter the following GraphQL document:

```
query find($search: String!) {
  findTrips(searchFor: $search) {
    id title startTime priority duration
  }
}
```

Here, we are declaring that our document takes a variable, $search, which is a String! (non-null string) value. To supply that value, we need to supply a map of variables, through a snippet of JSON.

Below the document pane, you should see a "QUERY VARIABLES" label. Below that will be another text editor — if "QUERY VARIABLES" is docked at the bottom of the browser window, click on it to expose the editor.

In that editor, type:

```
{
  "search": "ca"
}
```

If you now run the query in the document, you will see the same result as when we hard-coded the search criteria into the document itself.

Mutations and Objects

A mutation looks a lot like a query, in that it is a GraphQL operation that returns a

result set. However:

- It uses the `mutation` keyword
- Usually, a mutation will affect data on the server

So, for example, you might have:

```
mutation createTrip($trip: TripInput!) {
  createTrip(trip: $trip) {
    id
  }
}
```

Here, we have a GraphQL document with a `createTrip` mutation, asking the server to invoke `createTrip` and return an `id` of the presumably-created trip. Here, we use arguments and variables, but rather than using something simple like `String`, we have a `TripInput` value. `TripInput` is an "input object": a collection of named values representing the data to be inserted.

However, this particular GraphQL document will not work on the public demo GraphQL server and its GraphiQL tool. Because that server is public and does not require any authentication, it does not support modifying data, because then people might modify the data in ways that the author of this book might not like. In the next chapter, we will set up a local test server that you can use that supports mutations.

Errors

If we make a mistake, and execute a GraphQL document that is invalid for one reason or another, we will get errors back.

Our JSON responses from the demo server have all been in the form of a JSON object with a `data` property containing our results. If there are errors, though, we will get an `errors` property with the details.

For example, execute the following GraphQL document in the demo server's GraphiQL tool:

```
query find {
  findTrip(searchFor: "ca") {
    id title startTime priority duration
  }
```

26

```
}
```

This is the same as the hard-coded argument example from above, but with a typo: it has `findTrip` (singular) instead of `findTrips` (plural).

Executing this will result in errors:

```
{
  "errors": [
    {
      "message": "Cannot query field \"findTrip\" on type \"Query\". Did you mean
\"findTrips\" or \"getTrip\"?",
      "locations": [
        {
          "line": 2,
          "column": 3
        }
      ]
    }
  ]
}
```

Here, we get an error message, plus a location of where the error occurs within the original document. A development tool — like GraphiQL — might use this information to show you exactly where your mistake lies. Other types of clients might simply log these errors the same way that you might log exceptions.

GraphQL Test Environments

The GraphQL server that we have been using – `https://graphql-demo.commonsware.com/0.3/graphql` — only supports queries, off of a read-only data set. This prevents trolls from going in and adding... "interesting" data to the server, for some definition of "interesting".

However, that server lacks any support for mutations, and those are a critical piece of most GraphQL environments. And while it is set up to demonstrate a range of GraphQL capabilities, it does not cover everything.

In this chapter, we will get you set up with two additional environments for trying out GraphQL:

- a local test server, based on the same code that is used for the public demo server mentioned above
- GitHub, which has an extensive GraphQL API

The Test Server

Not only can you access the GraphQL demo server mentioned at the outset of this chapter, but also you can run the same server locally in your environment, such as on your development machine. The advantage is that the local copy *can* support mutations.

Setting up a local test server is optional. For example, IT rules may prohibit you from installing such a piece of software. However, that will mean that sample code that performs mutations will not work for you — you will be restricted to those samples that only perform queries.

Our Local Server: Express

There are a variety of server frameworks for implementing GraphQL endpoints. At this point, it is fairly likely that there is some server framework available in a programming language that is comfortable for you. The GraphQL Web site lists [many libraries and tools for GraphQL](#), many of which represent server frameworks. C. T. Lin's [awesome-graphql GitHub repo](#) lists even more.

The test server used in this book is implemented using [`express-graphql`](#). This is the reference implementation of a GraphQL server, maintained by the core GraphQL team. `express-graphql` is written in JavaScript, designed to be used by [Node.js](#) and [Express](#).

Since this is a "canned" server — you do not need to understand its implementation to use it — you should be able to set up and run the server even if you have limited experience with JavaScript.

Where to Run the Server

The simplest solution is to run the test server on your development machine. The test server is designed to be started and stopped from the command line, so you can run the server when you need it for testing and stop it when you no longer need it.

The key is that Android environments running your client code need to be able to reach the server. Depending on your network setup, that may or may not work with your development machine. For example, it may be that you are using a wired desktop, and it is on a separate LAN segment than is the WiFi that your test Android devices use.

`express-graphql` and its dependencies — including Node.js — are relatively lightweight. You can run the server on a wide range of environments, including a Raspberry Pi running Linux. So, if network restrictions prevent you from running the server on your development machine, you may be able to "get creative" about where your server code runs.

What you do *not* want to do is take this test server and put it on the public Internet, unless you are willing to take on the burdens of ensuring that only authorized parties can somehow get to that server.

Installing the Server

Setting up the test server is not especially difficult… once you get Node.js going.

Install Node.js

Node.js, if you are not familiar with it, is a framework for using JavaScript as an ordinary programming language, not just something that is used by Web browsers. In particular, Node.js is used for the server side of Web apps, allowing for a "full stack developer" to stick to a single programming language. Node.js uses Google's V8 engine for its JavaScript environment.

The test server has been tested with Node.js 6.11.0, which is on the long-term support (LTS) update track. It should work with anything in the 6.x versions, and it may work with newer versions of Node.js as well, though this has not been tested.

There are many different possible ways to install Node.js, depending on your operating system, whether you want multiple versions of Node.js to be available, whether you are deploying to a container (e.g., Docker) or not, and so on. This is not a Node.js book, and the author of this book is *so* not a Node.js expert. The [Node.js downloads page](#) is a good starting point for learning how to install Node.js.

When you are done, you should have `node` and `npm` commands available to you from the command line.

Obtain the Server Scripts

The test server consists of three files, both in the `Trips/Local/` directory of [the book's git repo](#):

- [`server.js`](#)
- [`server-v0.1.js`](#)
- [`package.json`](#)

The first two represent the actual server itself; the `package.json` file provides information about the dependencies on which those JavaScript files rely.

Download all three of these to a clean directory on your development machine or wherever you are planning on running the server.

These files will get updated from time to time. If you get an update to this book, be sure to update to the corresponding version of these files. The links shown above are to the `master` branch. Each book update's code in the git repo will be tagged with an equivalent version number (e.g., version 1.0 of the book results in a `1.0` tag in the repo), so you can download the right code to match your book version. There will also be an additional file per release, with the version number embedded in the filename (e.g., `server-v0.1.js`), with the particular GraphQL server logic that corresponds with that edition of the book.

Install Dependencies

The test server script relies on several dependencies. The `package.json` file lists them, along with compatible-version information, a bit reminiscent of how the `dependencies` closure works in Gradle.

Node.js, however, does not automatically download dependencies, any more than running an Android app downloads its dependencies. There is no compilation step with Node.js and JavaScript, though, so the dependency-download step is not handled there, the way it is with Gradle and Android apps.

So, you need to run a single command to download those dependencies:

```
npm install --save
```

That will create a `node_modules/` directory alongside the `server.js` and `package.json` files. In there, you will see a *long* list of directories representing Node.js packages (`npm` is the Node package manager). Some will be the dependencies listed in `package.json`; the rest will be transitive dependencies (the dependencies of your dependencies).

Running the Server

At this point, running the server is merely a matter of running `node server.js` from the directory where you downloaded `server.js` and ran the `npm install --save` command.

You should see a message like `Running a GraphQL API server at localhost:4000`. Press Ctrl-C to stop the server.

The port number — 4000 — is in the `server.js` code, towards the bottom, in case you need to change it to some other value.

One command-line switch is offered: `--hosted`. The presence or absence of this switch controls your starting data and mutation support:

Switch	Starting Data	Supports Mutations?
`--hosted`	yes	no
none	none	yes

The examples in the book that use this test server assume that you are running without the `--hosted` switch, and so there will be no starting data, but mutations are available.

Testing with GraphiQL

At this point, you can bring up `http://localhost:4000/0.3/graphql` in a Web browser and see a copy of GraphiQL for your local copy of the test server:
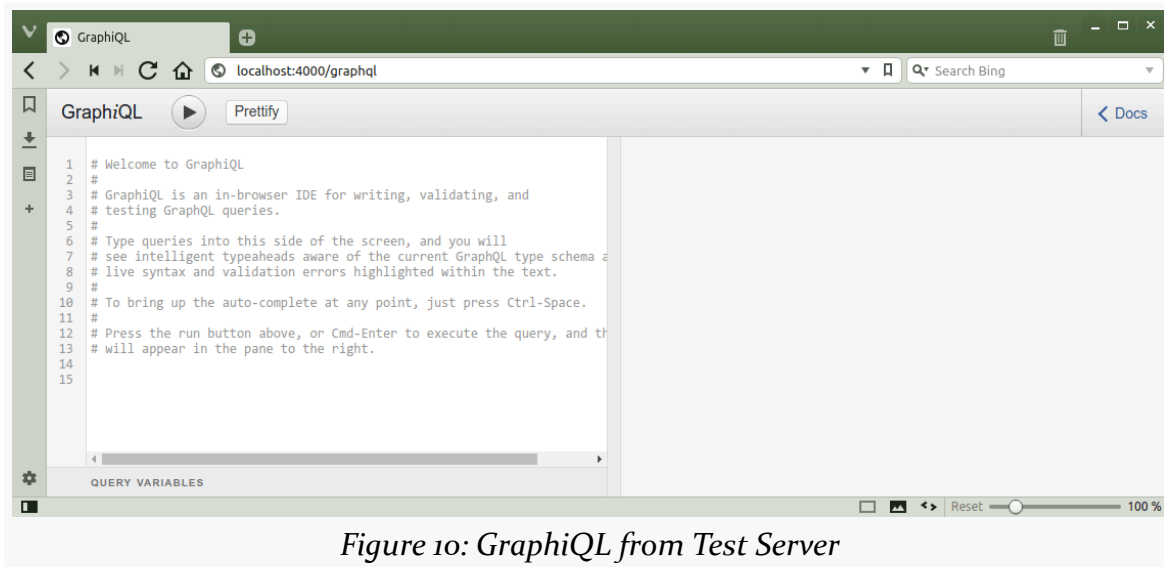
Figure 10: GraphiQL from Test Server

And, if you run some simple query, you will see that you have no data:
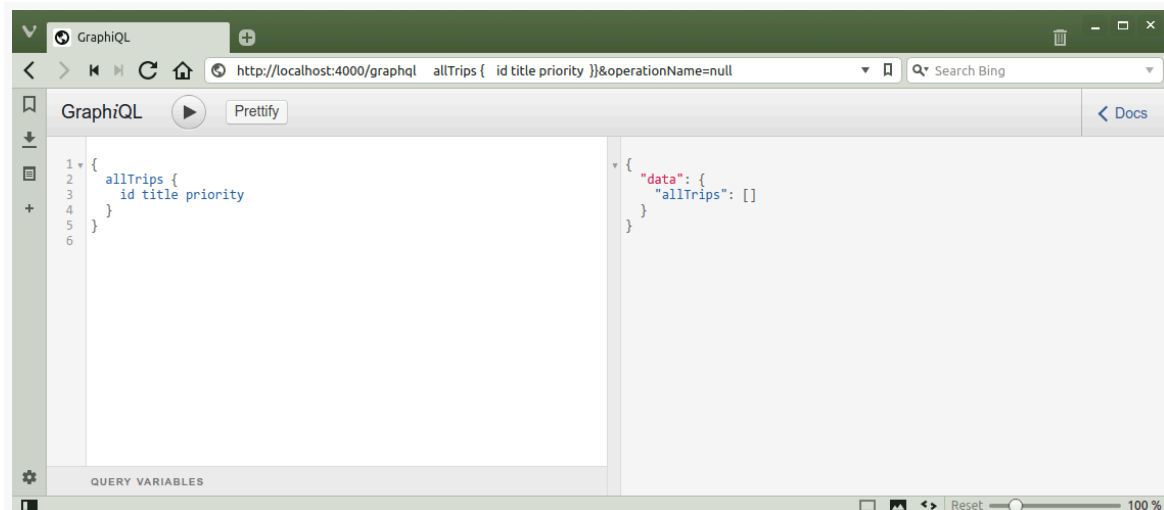
Figure 11: GraphiQL from Test Server, With Empty Results

When we start working with mutations, we will switch to your test server, so you can see how mutations work. Note, though, that the server does not persist any of the data that you submit — it is merely held in RAM. When you stop and start the server, you wipe out any previous mutation results.

In other words, do not store the travel plans for your real upcoming vacation in this server.

The `0.3/` path segment corresponds with this version of the book. Future versions of the book will have their own GraphQL logic and corresponding paths.

Trying a Mutation

Part of what we gain by having the local test server is the ability to perform mutations.

So, in the GraphiQL tool for your local test server, enter the following GraphQL document:

```
mutation createTrip($trip: TripInput!) {
  createTrip(trip: $trip) {
    id
  }
}
```

34

If you try to run that, you will get an error:

```
{
  "errors": [
    {
      "message": "Variable \"$trip\" of required type \"TripInput!\" was not
provided.",
      "locations": [
        {
          "line": 1,
          "column": 21
        }
      ]
    }
  ]
}
```

The mutation is expecting some input, and we did not provide it.

So, in the "QUERY VARIABLES" textarea, enter in the following JSON:

```
{
  "trip": {
    "startTime": "2017-05-03",
    "title": "A test trip",
    "priority": "LOW",
    "duration": 86400
  }
}
```

Running the document now should give you a UUID back, though your value will (hopefully) differ from:

```
{
  "data": {
    "createTrip": {
      "id": "1924f3bf-75d8-4e5b-8d26-bf6c40e5425c"
    }
  }
}
```

And now, if you try querying the server, using a document like:

```
{
  allTrips {
    id
```

```
    title
    startTime
    priority
    duration
  }
}
```

you will see your newly-added trip:

```
{
  "data": {
    "allTrips": [
      {
        "id": "1924f3bf-75d8-4e5b-8d26-bf6c40e5425c",
        "title": "A test trip",
        "startTime": "2017-05-03",
        "priority": "LOW",
        "duration": 86400
      }
    ]
  }
}
```

GitHub

One of the most interesting firms to offer a public GraphQL API is [GitHub](#), the popular project hosting service. GitHub has long had a REST-style Web service API, and in 2017 they added [a GraphQL API](#).

Getting an Authentication Token

To work with GitHub's GraphQL API, you first need to sign up for early-access program. This will require that you have a valid GitHub account and agree to a pre-release terms of service.

Then, you need to [create a Web service personal access token](#). The GitHub documentation focuses on this as being for command-line use, though in truth it gets used elsewhere. Creating one of these tokens requires you to select "scopes" that describe what data you want to have access to — in production, you would create a token that had as little access as possible, to limit the amount of possible damage should that token become publicly accessible. GitHub's [documentation](#) outlines a recommended set of scopes; those will be more than sufficient for your use with this book.

Accessing GitHub's GraphiQL

At that point, you can visit [GitHub's copy of GraphiQL](). When you do that, you will need to sign in with your GitHub account, in case you are using GraphiQL from a Web browser that is not already set up for use with GitHub.

And, with that, you can explore GitHub's GraphQL support, much in the same way that you explored the GraphQL demonstration site that this book references:
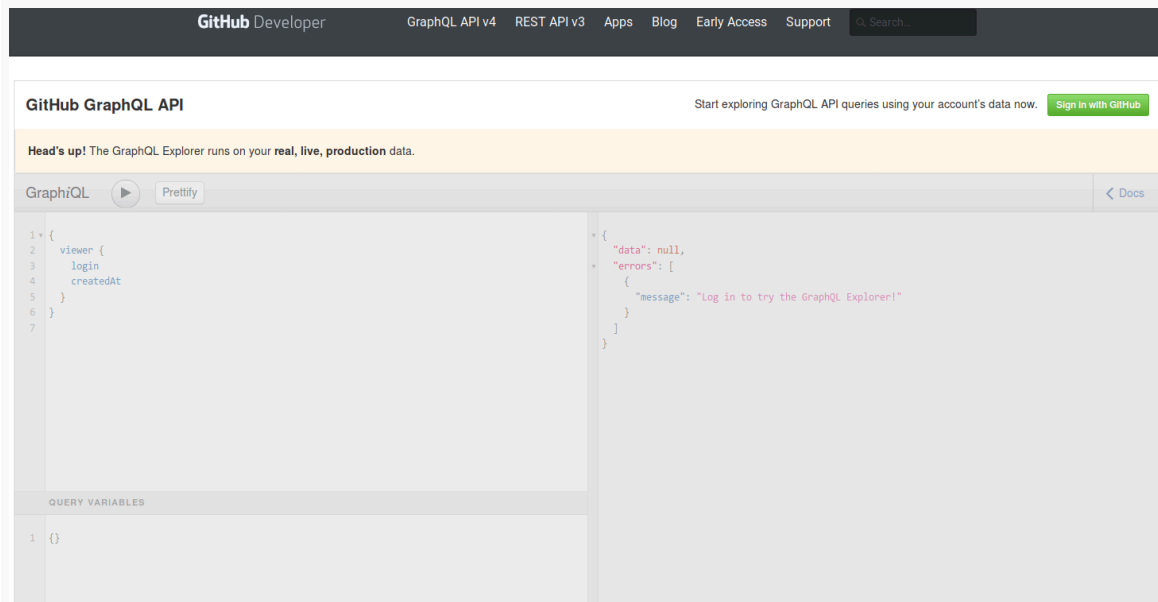
Figure 12: GitHub GraphiQL

GitHub API Documentation and Support

The GitHub Developer Web site has [extensive documentation of their API]().

GitHub also has [a Discourse site category]() for the GraphQL API, should you encounter any specific challenges in using it.

And Now, Onwards and Upwards!

Now that we have played around with GraphQL a bit, "gotten our feet wet", and have more test environments to play with, let's start playing around with GraphQL from Android apps.

Basic Dynamic GraphQL in Android

Now, let's see how we can start working with GraphQL from our own Android app.

Dynamic vs. Static

This chapter's title refers to "dynamic" GraphQL. And, if you peek ahead to the next chapter, you will see that it refers to "static" GraphQL. That distinction drives the libraries and tools that we can use for making our GraphQL requests.

Dynamic

Sometimes, the GraphQL request that we want to execute is not knowable when we write our app. GraphiQL, for example, allows you to enter in arbitrary GraphQL documents and execute operations from them.

In these cases, we need some code that can take the GraphQL, along with details like the server URL, and execute requests.

Static

Sometimes, the GraphQL representing our request(s) is known when we write our app. Pieces of data that go along with the request might not be known until runtime, though. For example, we might know that we want to find a trip based on a search term, but we do not know the search term until our app's user types one in on their device.

This is reminiscent of a SQL SELECT statement that we hard-code into our app, but some of the values to use in the WHERE clause are based on user input or other runtime values. In this case, we typically use positional parameters (e.g., SELECT *

FROM foo WHERE ID=?), so our SQL SELECT statement can still be hard-coded, but where we provide the parameter values at runtime.

As we saw in a previous chapter, GraphQL offers a similar separation, where we use a separate "variables" bit of JSON for the values that might vary at runtime. This allows us to have our GraphQL document as a plain, unvarying string.

The nice thing about having a GraphQL document as an ordinary string is that if that string is in a nice location — say, a plain text file in our project — we can do more work *at compile time* to make it easier for us to use that GraphQL.

We will explore all of this and more in the next chapter. The key for now is:

- "Dynamic" means "possibly not knowable until runtime"
- "Static" means "should be knowable at compile time"

GraphQL and HTTP

The GraphQL specification does not require any particular transport mechanism (e.g., HTTP) or particular serialization format (e.g., JSON). HTTP and JSON are *common* approaches, but they are not required.

Any time that you have a specification that does not specify something, you run into potential problems with multiple incompatible implementations of that "something".

While the GraphQL *specification* is agnostic, the GraphQL *documentation* does make some recommendations for how GraphQL-over-HTTP should work.

Of note, GraphQL usually uses a single URL or "endpoint" for all requests (e.g., http://graphql-demo.commonsware.com/0.3/graphql). This is in contrast with REST, which uses URLs as part of the content identification strategy.

A GraphQL request is made up of three pieces. The big one is the GraphQL document, containing your queries and mutations, written in GraphQL syntax. This is required for all requests. The others are:

- the variables (where needed), and
- the operation name, if there is more than one query or mutation in the document, to indicate which of those we really want to execute

You can use either an HTTP `GET` or `POST` to make the GraphQL request. A `GET` will use query parameters for the document, variables, and operation name:

Element	Required?	Query Parameter Name	Encoding
Document	yes	`query`	GraphQL, URL-encoded
Operation Name	sometimes	`operationName`	URL-encoded
Variables	sometimes	`variables`	JSON object, URL-encoded

The `operationName` *is* required if there are 2+ operations in the `document`, but if the `document` contains only a single operation, the `operationName` is not required. Similarly, if the requested operation takes variables, values for those variables are required; otherwise, the `variables` query parameter can be skipped.

Usually, a `POST` will wrap those three elements in JSON and deliver that as the payload on the request, with JSON object keys matching the query parameter names:

```
{
  "query": "query something($var1:String!, $var2:Int) { ... } ...",
  "operationName": "something",
  "variables": { "var1": "foo", "var2": 5 }
}
```

The MIME type of the `POST` request would be `application/json` in this case.

The HTTP response should be in the form of a JSON object as we have seen in the GraphiQL examples so far, containing `data` and/or `errors`.

Using OkHttp for GraphQL

You can use your favorite HTTP client API for executing GraphQL requests. In the Trips/CW/DynamicOk sample project, we use OkHttp to make a GraphQL request of the `graphql-demo.commonsware.com` server — the one that we used for GraphiQL earlier in the book. Specifically, we are going to run the same sort of `allTrips`

request as we did with GraphiQL:

```
{
  allTrips {
    id title startTime priority duration creationTime
  }
}
```

Note that the list of fields that we are to retrieve for the query here are separated by spaces, whereas some previous examples had them separated by newlines (plus spaces or tabs for indentation). Either works — from a specification standpoint, GraphQL requires that they be separated by some amount of whitespace.

In this specific sample, we are just going to dump the response JSON to a `TextView`, though in later samples, we will work on parsing the response into something more usable.

The Activity

Our `MainActivity`, which serves as this app's launcher activity, simply commits a `FragmentTransaction` to display a `SimpleTripsFragment`:

```java
package com.commonsware.graphql.trips.simple;

import android.app.Activity;
import android.os.Bundle;

public class MainActivity extends Activity {
  @Override
  public void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);

    if (getFragmentManager().findFragmentById(android.R.id.content) == null) {
      getFragmentManager().beginTransaction()
                      .add(android.R.id.content,
                          new SimpleTripsFragment()).commit();
    }
  }
}
```

(from [Trips/CW/DynamicOk/app/src/main/java/com/commonsware/graphql/trips/simple/MainActivity.java](#))

The UI

The UI for this sample app will be just a really big `TextView`, wrapped in a `ScrollView`, to show the JSON output:

```xml
<?xml version="1.0" encoding="utf-8"?>
<ScrollView xmlns:android="http://schemas.android.com/apk/res/android"
  android:layout_width="match_parent"
  android:layout_height="match_parent">

  <TextView
    android:id="@+id/result"
    android:layout_width="match_parent"
    android:layout_height="wrap_content"
    android:textSize="16sp"
    android:padding="8dp"
    android:typeface="monospace" />
</ScrollView>
```

(from Trips/CW/DynamicOk/app/src/main/res/layout/main.xml)

That layout is inflated in `onCreateView()` of `SimpleTripsFragment`, following a typical fragment setup process:

```java
@Override
public View onCreateView(LayoutInflater inflater, ViewGroup container,
                         Bundle savedInstanceState) {
  return(inflater.inflate(R.layout.main, container, false));
}
```

(from Trips/CW/DynamicOk/app/src/main/java/com/commonsware/graphql/trips/simple/SimpleTripsFragment.java)

The Dependencies

This app has four dependencies. Two, OkHttp and Gson, are tied to working with the GraphQL request and response itself. The other two, RxJava and RxAndroid, are for arranging to do the GraphQL work on a background thread and show the results on the main application thread:

```
dependencies {
  compile 'com.squareup.okhttp3:okhttp:3.5.0'
  compile 'com.google.code.gson:gson:2.4'
  compile 'io.reactivex.rxjava2:rxjava:2.0.2'
  compile 'io.reactivex.rxjava2:rxandroid:2.0.0'
}
```

If you are new to Rx, RxJava is the "reactive extensions for Java", providing an API for working with "streams" of data in a reactive fashion. As the data comes in, your code modifies and consumes that data. Many of the samples in this book will use RxJava and RxAndroid, and this book is not going to attempt to explain all the details of how those libraries work, as it is not central to using GraphQL in Android. You may wish to read more about Rx, whether from the author's *The Busy Coder's Guide to Android Development* or from another book that covers RxJava and RxAndroid.

The Query

We have a `query()` method that executes our GraphQL request and returns the raw JSON as a `String`:

```java
private String query() throws IOException {
  HashMap<String, String> payload=new HashMap<>();

  payload.put(QUERY, DOCUMENT);

  String body=new Gson().toJson(payload);
  Request request=new Request.Builder()
    .url(ENDPOINT)
    .post(RequestBody.create(MEDIA_TYPE_JSON, body))
    .build();
  Response response=new OkHttpClient().newCall(request).execute();

  return(response.body().string());
}
```

Here, we create a `HashMap` that is going to be used with an HTTP `POST` request to submit our GraphQL document to the server. Per the GraphQL documentation, we should create a JSON object with a `query` key that has our GraphQL document. In this case, the document is a simple `static` string, cunningly named `DOCUMENT`:

```java
private static final String DOCUMENT=
  "{ allTrips { id title startTime priority duration creationTime } }";
```

We then convert that `HashMap` to JSON via Gson, and wrap that JSON in a `RequestBody`, which is the OkHttp means of setting up a payload to be delivered with an HTTP request. We also need to provide a `MediaType` when creating the

RequestBody, to indicate the MIME type and encoding of the body. That `MediaType` is defined as a constant, `MEDIA_TYPE_JSON`:

```
private static final MediaType MEDIA_TYPE_JSON
  =MediaType.parse("application/json; charset=utf-8");
```

(from Trips/CW/DynamicOk/app/src/main/java/com/commonsware/graphql/trips/simple/SimpleTripsFragment.java)

Then, we build an OkHttp `Request`, supplying that `RequestBody` to the `post()` method (indicating that we want an HTTP `POST` request), plus providing the URL for the request, defined as an `ENDPOINT` constant:

```
private static final String ENDPOINT=
  "https://graphql-demo.commonsware.com/0.3/graphql";
```

(from Trips/CW/DynamicOk/app/src/main/java/com/commonsware/graphql/trips/simple/SimpleTripsFragment.java)

Then, we tell OkHttp to execute the `Request` and get the `Response`, returning the result as a `String`. If there is some problem (e.g., server is down), we throw an `IOException`.

So, when we call `query()`, we synchronously get back a `String` with the JSON results.

The Observable Chain

Up in `onCreate()`, we create an RxJava `Observable` based on `query()`, with a few operations chained onto it:

```
@Override
public void onCreate(Bundle savedInstanceState) {
  super.onCreate(savedInstanceState);

  setRetainInstance(true);

  observable=Observable
    .defer(new Callable<ObservableSource<String>>() {
      @Override
      public ObservableSource<String> call() throws Exception {
        return(Observable.just(query()));
      }
    })
    .subscribeOn(Schedulers.io())
    .map(this::prettify)
    .observeOn(AndroidSchedulers.mainThread())
```

```
    .cache();
}
```

Specifically, we:

- Use `defer()` and `Observable.just()` to wrap our method in an `Observable` that will invoke our `query()` method when we subscribe to it
- Request to use RxJava's `Schedulers.io()` thread for subscribing, meaning that our `query()` method will be called on that thread, keeping it off of the Android main application thread
- Convert the JSON that we get in via a `prettify()` method
- Request to use the main application thread for observing the results of the subscription and conversion, via `observeOn(AndroidSchedulers.mainThread())`, and
- Cache the results of the subscription

This `Observable` is held in a field of the fragment (`observable`), and the fragment is being retained. We are creating this `Observable` when the fragment is first created, so on a configuration change, while we wind up with a new `TextView`, we still have the original `Observable`. That, plus `cache()`, means that the results of our work will be retained across configuration changes, though not across process termination (as we are not saving this information in the saved instance state `Bundle`).

`prettify()` pretty-prints the JSON. We take advantage of the fact that we have access to Gson, and Gson has a fairly simple recipe for making the JSON more readable:

```java
private String prettify(String raw) {
  Gson gson=new GsonBuilder().setPrettyPrinting().create();
  JsonElement json=new JsonParser().parse(raw);

  return(gson.toJson(json));
}
```

The Subscription

Eventually, we need to subscribe to the `Observable`, to trigger the GraphQL request and get the results. We do that in `onViewCreated()`, as then we know that we have a place to put those results when they arrive:

```java
@Override
public void onViewCreated(View v, Bundle savedInstanceState) {
  super.onViewCreated(v, savedInstanceState);

  ((TextView)v.findViewById(R.id.result)).setHorizontallyScrolling(true);

  unsub();
  sub=observable.subscribe(
    this::updateText,
    error -> Toast
      .makeText(getActivity(), error.getMessage(), Toast.LENGTH_LONG)
      .show()
  );
}
```

<div align="center">(from <u>Trips/CW/DynamicOk/app/src/main/java/com/commonsware/graphql/trips/simple/SimpleTripsFragment.java</u>)</div>

After chaining to the superclass, we call setHorizontallyScrolling(true) on the
TextView. Despite the name, it does not enable horizontal scrolling. However, it *does*
block automatic word-wrapping, helping us to retain the pretty-print formatting.

Then, we subscribe() to the Observable, holding on to the result (an instance of
Disposable) in a field. The results get passed to the updateText() method, which
retrieves the current TextView and puts the text in it:

```java
private void updateText(String text) {
  ((TextView)getView().findViewById(R.id.result)).setText(text);
}
```

<div align="center">(from <u>Trips/CW/DynamicOk/app/src/main/java/com/commonsware/graphql/trips/simple/SimpleTripsFragment.java</u>)</div>

If an exception was raised by the call to query(), that gets routed to a Toast.

However, before we subscribe() to the Observable, we call an unsub() method.
This is also called in onDestroy():

```java
@Override
public void onDestroy() {
  unsub();

  super.onDestroy();
}
```

<div align="center">(from <u>Trips/CW/DynamicOk/app/src/main/java/com/commonsware/graphql/trips/simple/SimpleTripsFragment.java</u>)</div>

In unsub(), if we have an outstanding subscription and it has not already been

<div align="center">**47**</div>

disposed, we dispose of it:

```java
private void unsub() {
  if (sub!=null && !sub.isDisposed()) {
    sub.dispose();
  }
}
```

(from Trips/CW/DynamicOk/app/src/main/java/com/commonsware/graphql/trips/simple/SimpleTripsFragment.java)

This way, we dispose of our subscription:

- when the fragment is finally destroyed (e.g., user presses BACK to exit the activity) and `onDestroy()` is called, and
- before creating a fresh subscription to tie the results to our new widgets in our retained fragment

The Results

If you run the app, on a device or emulator that can connect to the server, you will see the results of querying for the `allTrips` field:
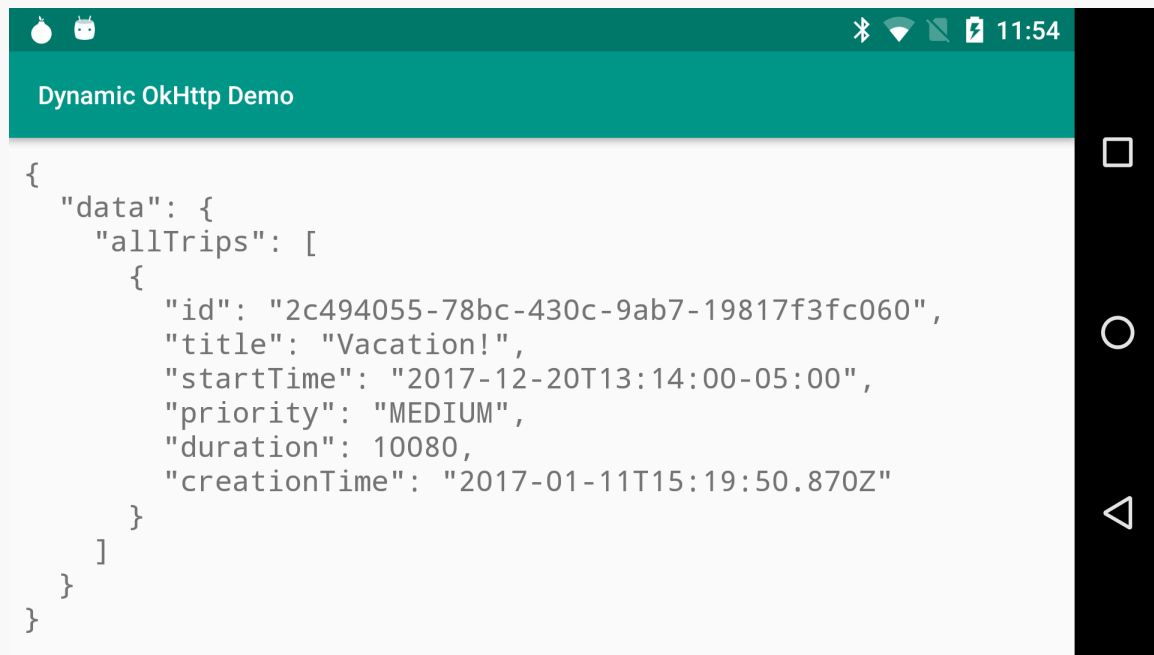
Figure 13: OkHttp Demo, Showing Some Results

Getting a Parsed Response

Showing the raw JSON may be fine for developers, but it is not a great choice for ordinary people. We would be better served showing the results in something like a list. And, for that, we really could use the results in a parsed form, rather than some string of JSON data.

The [Trips/CW/DynamicList](#) sample project does just that, using a `RecyclerView` for displaying the trips.

The Response Objects and Revised Query

A GraphQL response, as seen above, is a JSON object with `data` and possibly `error` properties.

So, to model that, we have a `GraphQLResponse` object that matches the GraphQL specification:

```java
package com.commonsware.graphql.trips.simple;

import java.util.ArrayList;
import java.util.HashMap;
import java.util.List;
import java.util.Map;

class GraphQLResponse {
  final Map<String, Object> data=new HashMap<>();
  final List<ResponseError> errors=new ArrayList<>();
}
```

<div align="center">(from Trips/CW/DynamicList/app/src/main/java/com/commonsware/graphql/trips/simple/GraphQLResponse.java)</div>

The `data` is simply a parsed representation of the JSON data as basic Java objects (lists, maps, strings, etc.).

The `errors`, in GraphQL, are a list of JSON objects containing an error message and a list of locations where that error appears. These are modeled by `ResponseError` and `ResponseError.Location` objects, respectively:

```java
package com.commonsware.graphql.trips.simple;

import java.util.ArrayList;
import java.util.List;
```

```java
class ResponseError {
  final String message=null;
  final List<Location> locations=new ArrayList<>();

  @Override
  public String toString() {
    return(message);
  }

  static class Location {
    final int line;
    final int column;

    Location() {
      line=-1;
      column=-1;
    }
  }
}
```

(from Trips/CW/DynamicList/app/src/main/java/com/commonsware/graphql/trips/simple/ResponseError.java)

We will be using Gson to parse this JSON. Gson handles `final` fields well overall, but we cannot use an initializer on primitives. Hence, `Location` initializes its `final` fields in a constructor, so that Gson can still revise them using Java reflection.

Now, `query()` can return a `GraphQLResponse`, provided to us by Gson:

```java
private GraphQLResponse query() throws IOException {
  HashMap<String, String> payload=new HashMap<>();
  Gson gson=new Gson();

  payload.put(QUERY, DOCUMENT);

  String body=gson.toJson(payload);
  Request request=new Request.Builder()
    .url(ENDPOINT)
    .post(RequestBody.create(MEDIA_TYPE_JSON, body))
    .build();
  Response okResponse=new OkHttpClient().newCall(request).execute();

  return(gson.fromJson(okResponse.body().charStream(), GraphQLResponse.class));
}
```

(from Trips/CW/DynamicList/app/src/main/java/com/commonsware/graphql/trips/simple/SimpleTripsFragment.java)

The Revised Rx and UI

This project no longer bothers with its own layout file. Instead, it uses a

RecyclerViewFragment, that roughly fills the same role that ListFragment does, except that it wraps a RecyclerView:

```java
package com.commonsware.graphql.trips.simple;

import android.app.Fragment;
import android.os.Bundle;
import android.support.v7.widget.RecyclerView;
import android.view.LayoutInflater;
import android.view.View;
import android.view.ViewGroup;

public class RecyclerViewFragment extends Fragment {
  @Override
  public View onCreateView(LayoutInflater inflater, ViewGroup container,
                           Bundle savedInstanceState) {
    RecyclerView rv=new RecyclerView(getActivity());

    rv.setHasFixedSize(true);

    return(rv);
  }

  public void setAdapter(RecyclerView.Adapter adapter) {
    getRecyclerView().setAdapter(adapter);
  }

  public RecyclerView.Adapter getAdapter() {
    return(getRecyclerView().getAdapter());
  }

  public void setLayoutManager(RecyclerView.LayoutManager mgr) {
    getRecyclerView().setLayoutManager(mgr);
  }

  public RecyclerView getRecyclerView() {
    return((RecyclerView)getView());
  }
}
```

(from Trips/CW/DynamicList/app/src/main/java/com/commonsware/graphql/trips/simple/RecyclerViewFragment.java)

Our SimpleTripsFragment extends from this RecyclerViewFragment, and so we inherit the RecyclerView.

The Observable that we create in onCreate() is unchanged, other than declaring the field to be an Observable of GraphQLResponse, rather than of String.

51

onViewCreated(), though, needs to set up the RecyclerView plus subscribe to that Observable:

```java
@Override
public void onViewCreated(View v, Bundle savedInstanceState) {
  super.onViewCreated(v, savedInstanceState);

  setLayoutManager(new LinearLayoutManager(getActivity()));

  getRecyclerView()
    .addItemDecoration(new DividerItemDecoration(getActivity(),
      LinearLayoutManager.VERTICAL));

  unsub();
  sub=observable.subscribe(
    response -> setAdapter(buildAdapter(response)),
    error -> {
      Toast
        .makeText(getActivity(), error.getMessage(), Toast.LENGTH_LONG)
        .show();
      Log.e(getClass().getSimpleName(), "Exception processing request",
        error);
    });
}
```

<div align="center">(from <u>Trips/CW/DynamicList/app/src/main/java/com/commonsware/graphql/trips/simple/SimpleTripsFragment.java</u>)</div>

Here, in subscribe(), when we get our results (response), we call the setAdapter() method inherited from RecyclerViewFragment, using a buildAdapter() method to build a RecyclerView.Adapter based on the SimpleResponse:

```java
private RecyclerView.Adapter buildAdapter(GraphQLResponse response) {
  if (response.errors!=null && response.errors.size()>0) {
    Toast
      .makeText(getActivity(), response.errors.get(0).toString(), Toast.LENGTH_LONG)
      .show();

    for (ResponseError error : response.errors) {
      Log.e(getClass().getSimpleName(), error.toString());
    }
  }

  List<Map<String, Object>> allTrips;

  if (response.data!=null) {
    allTrips=(List<Map<String, Object>>)response.data.get(KEY_ALL_TRIPS);
  }
  else {
    allTrips=new ArrayList<>();
  }
```

```
  return(new TripsAdapter(allTrips, getActivity().getLayoutInflater(),
    android.text.format.DateFormat.getDateFormat(getActivity())));
}
```

If we got `errors` in our response, we show the first error message via a `Toast` and log each of the errors to LogCat.

The `data` in our JSON response is a JSON object with a single key, `allTrips`. That key, in turn, points to a `List` of JSON objects representing the trips served by the server. So, we retrieve that `List` of `Map` objects, where `KEY_ALL_TRIPS` is a constant defined as `allTrips`. Then, we return a `TripsAdapter` wrapped around that, also providing it a `LayoutInflater` (to inflate rows) and a `DateFormat` for use in formatting date values that we get back from the JSON.

(the fully-qualified `android.text.format.DateFormat` is because we are also using Java's standard `DateFormat` class, and we cannot `import` both of them)

If we did not get any data (e.g., the query had errors), we use an empty `ArrayList` instead.

`TripsAdapter` wraps the `List` of `Map` objects and sets up `RowHolder` instances for each as needed, with rows defined using the Android SDK's `simple_list_item_1` layout:

```
private static class TripsAdapter extends RecyclerView.Adapter<RowHolder> {
  private final List<Map<String, Object>> trips;
  private final LayoutInflater inflater;
  private final DateFormat dateFormat;

  private TripsAdapter(List<Map<String, Object>> trips,
                       LayoutInflater inflater, DateFormat dateFormat) {
    this.trips=trips;
    this.inflater=inflater;
    this.dateFormat=dateFormat;
  }

  @Override
  public RowHolder onCreateViewHolder(ViewGroup parent,
                                      int viewType) {
    return(new RowHolder(inflater.inflate(android.R.layout.simple_list_item_1,
      parent, false), dateFormat));
  }

  @Override
  public void onBindViewHolder(RowHolder holder,
                               int position) {
    holder.bind(trips.get(position));
  }
```

53

```
  @Override
  public int getItemCount() {
    return(trips.size());
  }
}
```

(from Trips/CW/DynamicList/app/src/main/java/com/commonsware/graphql/trips/simple/SimpleTripsFragment.java)

RowHolder fills in the TextView for the row with the title and startTime values from the trip, obtained by fetching the appropriate values from the Map representing that trip. The startTime is in ISO8601 format, which is not very human-friendly, so we parse that into a Date and format it into a friendlier form using the aforementioned DateFormat object:

```
private static class RowHolder extends RecyclerView.ViewHolder {
  private final TextView rowLabel;
  private final DateFormat dateFormat;

  RowHolder(View itemView, DateFormat dateFormat) {
    super(itemView);

    rowLabel=(TextView)itemView.findViewById(android.R.id.text1);
    this.dateFormat=dateFormat;
  }

  void bind(Map<String, Object> trip) {
    String startTime=trip.get(KEY_START_TIME).toString();
    String title=trip.get(KEY_TITLE).toString();

    try {
      Date parsedStartTime=ISO8601.parse(startTime);
      rowLabel.setText(String.format("%s : %s",
        dateFormat.format(parsedStartTime), title));
    }
    catch (ParseException e) {
      Log.e(getClass().getSimpleName(), "Exception parsing "+startTime, e);
    }
  }
}
```

(from Trips/CW/DynamicList/app/src/main/java/com/commonsware/graphql/trips/simple/SimpleTripsFragment.java)

The Revised Results

Now, our results are formatted as a list of trips, though the list items do not show all the details of each trip:
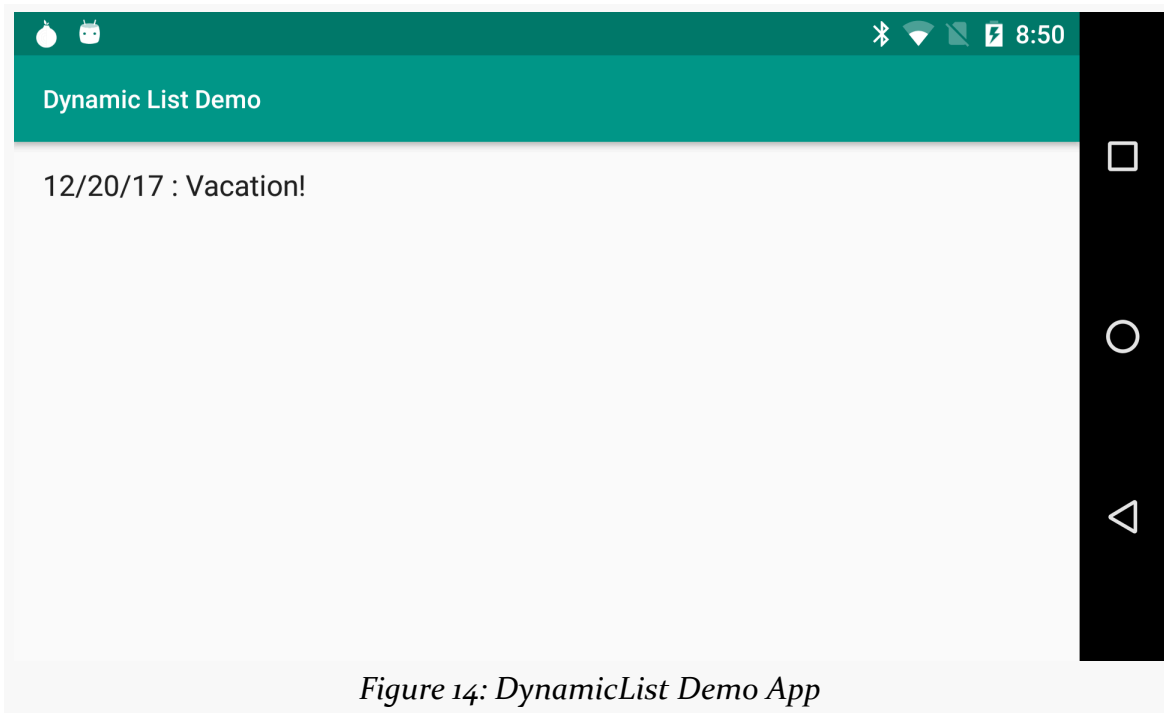
Figure 14: DynamicList Demo App

One could arrange to show the rest of the trip detail, such as by responding to clicks on the `RecyclerView` rows and bringing up a detail screen about the trip.

Can't We Do Better Than Maps of Objects?

`GraphQLResponse` is simple to invoke, but the data that you get back is rather cumbersome to use. It is just a `Map`, not a first-class POJO.

Typically, when you use a JSON parser like Gson or Moshi, you are asking it to bind data into POJOs of your own design, not stock collection classes like `Map`. So, for example, you could imagine a `Trip` class like:

```java
public class Trip {
  public String id;
  public String startTime;
  public String title;
```

```
  public List<Note> notes;
  public String creationTime;
  public String updateTime;
  public Priority priority;
  public Integer duration;
  public List<Plan> plans;
}
```

which in turn holds onto other custom POJOs, like `Note` and `Priority` and `Plan`.

You can certainly go that route, replacing the `Map` in `GraphQLResponse` with a `List` of `Trip` objects. However, this is only practical when your GraphQL is static, or at least "static enough" that you know the precise JSON structure that you get back. For truly dynamic GraphQL, you do not know in advance what you will get back, so the `SimpleResponse` "it's all based on `List` and `Map`" is as good as you are going to get. And, if your GraphQL *is* static, there is another solution, one that we will explore in the next chapter.

Basic Static GraphQL in Android

Dynamic GraphQL is great for those cases where you happen to need it. More often than not, though, static GraphQL will suffice, as typically you know ahead of time what sorts of data you will be retrieving and modifying through GraphQL.

In this chapter, we will explore how static GraphQL documents can be used by a code generator to give you Java classes that make requests of the server and model the responses, so you can work with the server via a generated Java API. This is in contrast to what we did in [the previous chapter](#), where we used general-purpose APIs (e.g., OkHttp) that were not tied directly to our server and data model.

Android Apps and Code Generation

Ahead-of-time code generation abounds in Android app development. Every Android app winds up with code-generated Java classes, simply as part of normal development. Many developers do not even think of this as "code generation", just because it becomes such a natural extension of the way that Android apps get developed.

In a typical Android Studio project, your module will have a `build/generated/` directory containing generated Java code. If you rummage through what's there, you will see varying amounts of code, depending on what Android SDK and third-party code generators you wind up employing.

R and BuildConfig

When you write an Android app, any time your Java code wants to refer to a resource, you wind up using some `R` value, such as `R.string.app_name` or `R.layout.main`. The `R` class gets code-generated by `aapt` as part of compiling an

Android app. `aapt` is part of the "build tools" (tied to the `buildToolsVersion`), and one of its jobs is to parse all of the resources, identify all of the necessary symbols, and generate the `R` class in the manifest-declared `package` for your use. In a typical Android Studio project, each module will have a `build/generated/source/r` directory containing your generated `R` class for each build variant (e.g., `debug/`).

Similarly, `BuildConfig` is created by the build tools based on your Gradle build script, with information like your version code, version name, build type, and so on. You can add your own information to the generated `BuildConfig` class, for all build types or by product flavor, using `buildConfigField`. This is a popular way of making API keys available to Java code, for example. In a typical Android Studio project, each module will have a `build/generated/source/buildConfig` directory containing your generated `BuildConfig` class, again with one for each build variant (e.g., `debug/`).

AIDL

If you work with remote services, frequently you wind up writing some [AIDL](#) files. These work like IDL files do for RPC systems, providing a language-neutral way of describing an interface between programs. In RPC, the interface is between a client and a server; with AIDL, the interface is between a service and its clients in other apps.

The AIDL that you write results in a code-generated Java class that provides:

- A stub implementation of that API for your service to use, and
- A client-side proxy for making that API appear to be a local object in the client app, where the proxy handles all of the details of getting the request from the client to the server via Android's IPC system

Even if you do not have any AIDL, your Android Studio module may still generate an empty `build/generated/source/aidl/` directory. If you *do* have AIDL, your generated Java classes will reside in there.

Data Binding

Google introduced the [data binding framework](#) in 2015, trying to make it simpler for you to populate a widget-based UI from your Java code. Given an appropriately-augmented layout resource, the data binding code generator creates corresponding Java classes that allow you to populate those widgets, using binding code expressions written in the layout files themselves. All the Java code needs to do is create an

instance of the binding (which also inflates the layout), then supply the objects to be bound to that binding — the generated code and associated library takes care of the rest.

Compiled SQL Queries

Code generation is not just for Google — other developers have gotten into the game.

For example, Square's [SQLDelight](#) takes a file of SQL code and code-generates Java classes that provide a data-access layer for working with SQLite using that SQL code. You add the code generator Gradle plugin to your project, add the SQL file, and SQLDelight takes over from there.

The Staticizer

Implementing your own code-generating Gradle plugin is not that difficult, particularly using Square's [JavaPoet](#) library. JavaPoet is a Java library for generating Java code, using a builder-style API, to minimize the odds of syntax errors and to handle a lot of the dirty work (e.g., adding appropriate `import` statements) for you.

[The Busy Coder's Guide to Android Development](#) has a chapter on code generation that demonstrates the use of JavaPoet and Gradle plugins. That chapter profiles a [Staticizer](#) sample code generator that takes a JSON object and creates a corresponding Java class file, akin to `BuildConfig`. The idea is that you might get that JSON as part of the build process (e.g., from a server), and so the code generation eliminates the need to manually update some Java source to match the new JSON data.

Introducing Apollo and Apollo-Android

A GraphQL client-side code generator would take a GraphQL document and generate a Java class that hides all of the HTTP and JSON stuff behind a fairly straightforward API. Along the way, it might create POJOs that mirror the results that we are requesting from the server, so that we can work with ordinary Java objects, rather than having to mess with a nested series of `Map` and `List` objects.

[Apollo-Android](#) is such a code generator. It is created as part of the [Apollo Data](#) suite of libraries and tools for GraphQL development. Apollo Data, in turn, is part of the [Meteor Development Group](#), creators of the Meteor Web development stack.

A typical Android Studio project would use `com.apollographql.apollo:gradle-plugin`, which is a Gradle plugin that adds the code generation tasks to the standard Android Studio and Gradle build process. Given that, what you need to do is:

- Set up a directory with your GraphQL document(s) in your module
- Possibly add some configuration to your module's `build.gradle` file

The Gradle plugin takes it from there, generating your Java classes for you, which you can then use to make requests of the GraphQL server.

This sounds easy. In truth, it is rather complicated to set up and get working.

NOTE: These are early days for the Apollo-Android project. As a result, while the instructions shown in this chapter are up to date for when this book version was published, Apollo-Android may have changed since then. You may encounter some issues when trying to follow these instructions. Future updates to this book will update these instructions to the then-current edition of Apollo-Android.

Using Apollo-Android

The Trips/CW/StaticList sample project is a clone of last chapter's `DynamicList` project, except this time, we use Apollo-Android and the generated API, rather than the combination of OkHttp and Gson.

Installing Node

Back in [the chapter on GraphQL test environments](), we had you install NodeJS. Partially, that was to run the Express Web server that drives our local GraphQL server.

But, partially, that was for use with Apollo-Android.

The Apollo team, at its core, focuses on Web development. They have a fair bit of JavaScript code related to GraphQL. They elected to reuse some of that as part of creating their code generator, presumably with an eye towards reuse. The cost is complexity, both for them as a development team (combining Java and JavaScript for the code generation) and for you as a developer (having to install and set up NodeJS to be able to build an Android app).

Regardless, if you skipped over [setting up NodeJS](), if you wish to use the static

GraphQL sample apps, you will need to go back and take care of that now.

Installing apollo-codegen

The specific piece of reusable NodeJS code that Apollo-Android depends upon is `apollo-codegen`. As the name suggests, this module knows how to generate code, in this case based upon GraphQL documents and schemas.

`apollo-codegen` is a Node package, akin to the Express Web server that our test server uses. However, in this case, we need to install it directly from the Node package repository.

To do that, you can run the `npm install -g apollo-codegen` command. This will install `apollo-codegen` "globally", so it is available to any Android project for which you wish to use Apollo-Android. Note, though, that using the `-g` switch for a global install may require superuser privileges (e.g., `sudo`) on your development machine. In principle, you might be able to do a local installation of `apollo-codegen` via `npm install apollo-codegen`, but it is unclear if Apollo-Android will be able to find it that way.

Setting the Gradle Version

Usually, we set the version of Gradle that our Android projects use to whatever version causes Android Studio to stop complaining about our version of Gradle. So, for example, Android Studio 2.2.3 is happy with Gradle 2.14.1.

However, Apollo-Android itself wants Gradle 3.3 or higher. That is set in the `gradle/wrapper/gradle-wrapper.properties` file in a project:

```
#Wed Apr 10 15:27:10 PDT 2013
distributionBase=GRADLE_USER_HOME
distributionPath=wrapper/dists
zipStoreBase=GRADLE_USER_HOME
zipStorePath=wrapper/dists
distributionUrl=https\://services.gradle.org/distributions/gradle-3.3-all.zip
```

(from [Trips/CW/StaticList/gradle/wrapper/gradle-wrapper.properties](#))

Adding the Plugin

The Apollo-Android plugin, like all Gradle plugins, gets loaded via the `buildscript` closure in the top-level `build.gradle` file in your Android project.

The `StaticList` project's `build.gradle` file not only loads that plugin, but it also sets up some repositories for use by all modules in the project:

```
buildscript {
    repositories {
        jcenter()
    }
    dependencies {
        classpath 'com.android.tools.build:gradle:2.3.3'
        classpath 'com.apollographql.apollo:gradle-plugin:0.4.0'
    }
}

allprojects {
    repositories {
        jcenter()
        maven { url 'https://maven.google.com' }
    }
}
```

(from Trips/CW/StaticList/build.gradle)

The `classpath` statement, loading `com.apollographql.apollo:gradle-plugin`, loads the Apollo-Android plugin, much as the preceding line loads the Android plugin.

With the Android plugin, we load it in the top-level `build.gradle`, but then we apply it in a module's `build.gradle` file, using the `apply plugin` statement (e.g., `apply plugin: 'com.android.application'`). Similarly, we need to have an `apply plugin` statement to apply the Apollo-Android plugin – `apply plugin: 'com.apollographql.android'`:

```
apply plugin: 'com.android.application'
apply plugin: 'com.apollographql.android'
```

(from Trips/CW/StaticList/app/build.gradle)

The combination of the `classpath` entry in the project `build.gradle` and the `apply plugin` statement gives your module Super GraphQL Code-Generating Powers™.

We also need to add the `apollo-rx2-support` artifact to our dependencies. As the name suggests, this adds RxJava/RxAndroid hooks that we can use with our Apollo-generated code. It and the plugin itself will also give us the main runtime libraries for Apollo. That gives us the following dependency list:

```
dependencies {
  compile 'io.reactivex.rxjava2:rxjava:2.0.2'
  compile 'io.reactivex.rxjava2:rxandroid:2.0.0'
  compile 'com.android.support:recyclerview-v7:26.0.1'
  compile 'com.apollographql.apollo:apollo-rx2-support:0.4.0'
}
```

(from Trips/CW/StaticList/app/build.gradle)

We can configure the plugin via an `apollo` closure. This would be a peer of the `android` closure, which configures the Android Plugin for Gradle. Most of the Apollo-Android projects in this book will have an `apollo` closure that looks like this:

```
apollo {
  useSemanticNaming = false
}
```

(from Trips/CW/StaticList/app/build.gradle)

The `useSemanticNaming` property that we are setting to `false` has an impact on the names of classes that Apollo-Android code-generates for us, as we will see later in this chapter.

Downloading the Schema

Next, we need to download some details from the GraphQL server that we plan to connect to. This is handled via an `apollo-codegen` command-line utility that was installed when you installed the `apollo-codegen` Node module.

In your module, create a `src/main/graphql/` directory. This is a magic location that Apollo-Android uses to find relevant files for the build process.

Inside there, create a `com/commonsware/graphql/trips/api` directory. This is a Java-style namespace directory tree, indicating the Java package that Apollo-Android should use for its code generation.

Then, from your module's root directory, run the following command:

```
apollo-codegen download-schema https://graphql-demo.commonsware.com/0.3/graphql --output src/main/graphql/com/commonsware/graphql/trips/api/schema.json
```

(NOTE: while that may word-wrap as you are reading this book, it should be typed on all one line)

This tells `apollo-codegen` to download a JSON file representing the server's GraphQL schema, storing that file in the directory that you just created. This uses [the introspection APIs](), much as how GraphiQL uses those APIs to validate your GraphQL documents. The resulting `schema.json` file will be placed into the `src/main/graphql/com/commonsware/graphql/trips/api/schema.json` directory that you created.

For projects that you get from this book's Git repo, you will not need to download the schema, as that file will be commited to the repo.

Writing the GraphQL

The `src/main/graphql/com/commonsware/graphql/trips/api/` directory serves two roles for Apollo-Android. First, it is where the `schema.json` file goes. Second, it is where your static GraphQL documents go.

For example, the `StaticList` project has a `com/commonsware/graphql/trips/api/` directory tree under `graphql/`, and in there has a `TripServer.graphql` file:

```
query getAllTrips {
  allTrips {
    id
    title
    startTime
    priority
    duration
    creationTime
  }
}
```

(from [Trips/CW/StaticList/app/src/main/graphql/com/commonsware/graphql/trips/api/TripServer.graphql]())

The Java code that Apollo-Android generates will wind up in the `com.commonsware.graphql.trips.api` Java package, based upon that directory tree. Note that while the filename needs the `.graphql` extension, the base name (`TripServer`) seems to be ignored at the present time.

What You Get

Each GraphQL operation, across the GraphQL documents, results in a Java class, in your designated package, that contains an API for that operation. The Java class name is the same as the name of the operation, normalized to Java class naming

conventions. So, the `getAllTrips` operation in the `StaticList` GraphQL document results in a `GetAllTrips` Java class.

That source file, and others, will be written into `build/generated/source/`, alongside the generated source for `R`, `BuildConfig`, any AIDL you may have, and so on. Specifically, inside of `build/generated/source/` will be:

- An `apollo/` directory, containing...
- a traditional set of Java package directories (e.g., `com/commonsware/graphql/trips/simple/api/`), containing...
- the generated Java code, such as `GetAllTrips.java`

The inner workings of the generated Java code are undocumented. Certain aspects of that class, though, surface themselves in the way that we use the class to access our GraphQL server. We will see more about `GetAllTrips`, therefore, as we review the code that replaces our dynamic GraphQL request with this static one from Apollo-Android.

The reason why the Java class is named `GetAllTrips` is that is what we are naming our query in the GraphQL. Had we set `useSemanticNaming` to `true` in that `apollo` closure in the `build.gradle` file, the Java class would be `GetAllTripsQuery`. Either way is fine; for simplicity and brevity, the book's sample app turn off this "semantic naming" feature.

Using the Generated APIs

Given all that setup, Apollo-Android is now in position to code-generate for us an API, tied to our specific GraphQL schema and document. It is up to us to apply that generated code in our app.

Creating an ApolloClient

The first step is to create an instance of `ApolloClient`. This is a wrapper around an `OkHttpClient` that knows about Apollo-Android and how to work with the generated code. To create such an instance, call `ApolloClient.builder()`, call various builder methods, and `build()` the resulting `ApolloClient` instance:

```
private ApolloClient apolloClient=ApolloClient.builder()
  .okHttpClient(new OkHttpClient())
  .serverUrl("https://graphql-demo.commonsware.com/0.3/graphql")
  .build();
```

At minimum, you need to call:

- okHttpClient(), to supply an OkHttpClient configured how you want
- serverUrl(), indicating the server that you want to access

If you need to work with more than one GraphQL server, you will need more than one ApolloClient instance, though perhaps all sharing a common OkHttpClient.

Making the Observable

The apollo-rx2-support dependency gives us access to the Rx2Apollo helper class, which has static methods to assist us in setting up RxJava Observable chains leveraging Apollo-Android's generated code:

```java
@Override
public void onCreate(Bundle savedInstanceState) {
  super.onCreate(savedInstanceState);

  setRetainInstance(true);

  observable=Rx2Apollo.from(apolloClient.query(new GetAllTrips()).watcher())
    .subscribeOn(Schedulers.io())
    .map(response -> (getAllTripsFields(response)))
    .cache()
    .observeOn(AndroidSchedulers.mainThread());
}
```

Here, we:

- Create an instance of GetAllTrips, the generated Java class representing the getAllTrips query from our GraphQL document
- Pass that GetAllTrips instance to newCall() on the ApolloClient, and call watcher() on that
- Pass the results of watcher() to from() on Rx2Apollo, which gives us an Observable on a Response from Apollo-Android

That Response has our actual results, in the form of a GetAllTrips.Data object, where the .Data nested class represents the results of executing a getAllTrips query. Calling data() on the Response will give us that GetAllTrips.Data object. And, if there are errors in the GraphQL, the Response will return an errors() list.

So, in the `Observable` chain, we `map()` the `Response` to a `GetAllTrips.Data` object via `getAllTripsFields()`:

```java
private GetAllTrips.Data getAllTripsFields(Response<GetAllTrips.Data> response) {
  if (response.hasErrors()) {
    throw new RuntimeException(response.errors().get(0).message());
  }

  return(response.data());
}
```

<div align="center">(from <u>Trips/CW/StaticList/app/src/main/java/com/commonsware/graphql/trips/simple/SimpleTripsFragment.java</u>)</div>

We also see if there are errors, and if there are we throw a `RuntimeException`, containing the first error's message, as a way to get the error message into the RxJava flow.

From there, we set up the RxJava/RxAndroid thread rules and `cache()` the results (to deal with configuration changes).

Working with the Results

Later on, in `onViewCreated()`, we `subscribe()` to the `Observable` as before, calling `buildAdapter()` to process the results of the query. Since our `Observable` is on `GetAllTrips.Data`, `buildAdapter()` now takes a `GetAllTrips.Data` object. Since our GraphQL query is retrieving the `allTrips` field, there is a generated `allTrips()` method on `GetAllTrips.Data` that returns a `List` of `GetAllTrips.AllTrip` objects. We can pass that `List` along to the `TripsAdapter`:

```java
private RecyclerView.Adapter buildAdapter(GetAllTrips.Data response) {
  return(new TripsAdapter(response.allTrips(),
    getActivity().getLayoutInflater(),
    android.text.format.DateFormat.getDateFormat(getActivity())));
}
```

<div align="center">(from <u>Trips/CW/StaticList/app/src/main/java/com/commonsware/graphql/trips/simple/SimpleTripsFragment.java</u>)</div>

`TripsAdapter` now manages this `List` of `GetAllTrips.AllTrip` objects, each of which represents a single trip. Similarly, our `RowHolder` can now bind instances of `GetAllTrips.AllTrip` to our rows:

```java
private static class TripsAdapter extends RecyclerView.Adapter<RowHolder> {
  private final List<GetAllTrips.AllTrip> trips;
  private final LayoutInflater inflater;
  private final DateFormat dateFormat;

  private TripsAdapter(List<GetAllTrips.AllTrip> trips,
```

<div align="center">**67**</div>

```
                      LayoutInflater inflater, DateFormat dateFormat) {
    this.trips=trips;
    this.inflater=inflater;
    this.dateFormat=dateFormat;
  }

  @Override
  public RowHolder onCreateViewHolder(ViewGroup parent,
                                      int viewType) {
    return(new RowHolder(inflater.inflate(android.R.layout.simple_list_item_1,
      parent, false), dateFormat));
  }

  @Override
  public void onBindViewHolder(RowHolder holder,
                               int position) {
    holder.bind(trips.get(position));
  }

  @Override
  public int getItemCount() {
    return(trips.size());
  }
}

private static class RowHolder extends RecyclerView.ViewHolder {
  private final TextView rowLabel;
  private final DateFormat dateFormat;

  RowHolder(View itemView, DateFormat dateFormat) {
    super(itemView);

    rowLabel=(TextView)itemView.findViewById(android.R.id.text1);
    this.dateFormat=dateFormat;
  }

  void bind(GetAllTrips.AllTrip trip) {
    try {
      Date parsedStartTime=ISO8601.parse(trip.startTime());
      rowLabel.setText(String.format("%s : %s",
        dateFormat.format(parsedStartTime), trip.title()));
    }
    catch (ParseException e) {
      Log.e(getClass().getSimpleName(), "Exception parsing "+trip.startTime(), e);
    }
  }
}
```

(from [Trips/CW/StaticList/app/src/main/java/com/commonsware/graphql/trips/simple/SimpleTripsFragment.java](#))

Names and Apollo-Android

To recap:

- The name of our static GraphQL document (e.g., `TripServer.graphql`) is ignored

- The names of the operations in that GraphQL document (e.g., `getAllTrips`) drives the name of the generated root Java classes (e.g., `GetAllTrips`)
- The names of the fields returned by the operation (e.g., `allTrips`) drives the name of the response classes (e.g., `GetAllTrips.AllTrip`) and the method that returns them (e.g., `allTrips()`)

Was All of This Worth It?

The idea behind code generation is to minimize boilerplate. In particular, with GraphQL, we want to have a single "source of truth" (GraphQL schema and documents) that powers our requests and the parsing of the responses.

However, there is a fair bit of work to get Apollo-Android set up. Many Android developers will have little experience with NodeJS, and the requirement for using it adds to the size and complexity of the overall project.

This setup work is a "fixed cost", more or less, independent of project size. As a result, larger projects are more likely to gain value from Apollo-Android than are smaller projects.

Exploring GraphQL Syntax

Objects, Fields, and Types

GraphQL queries and mutations do not follow syntax conventions that you are used to. In some respects, they look a bit like function calls. In other respects... well, GraphQL is strange.

That being said, its foundation lies in the same sorts of objects, fields, and data types that you may be used to from other forms of programming, such as Java. However:

- The objects are not really objects
- The fields are only sometimes what you think of as fields
- The data types are fairly normal, at least until you get to lists

In this chapter, we will explore how the objects, fields, and data types work in our GraphQL requests, and we will examine some of the idiosyncrasies involved with GraphQL.

Introducing the GraphQL Schema Definition Language

When we create databases in Android with SQLite, we define a schema by means of SQL statements like `CREATE TABLE`. When we define an bound service interface, we define a "schema" of sorts by means of AIDL, if we are delivering that service across process boundaries.

Similarly, with GraphQL, we can define the different types that a server exposes by means of [the GraphQL schema definition language](#).

That language is used on the server, not on the client. However, when describing GraphQL, often times it is useful to show snippets of GraphQL schema language, as

73

this is a compact way of depicting these types. Hence, while you may never wind up needing to *write* GraphQL schemas, it will be helpful if you can *read* them, which is why they are presented in this chapter.

This "cheat sheet" provides a capsule description of how the schema language looks and works.

Objects

Let's start by going back to the generated documentation in GraphiQL that we saw earlier in the book:
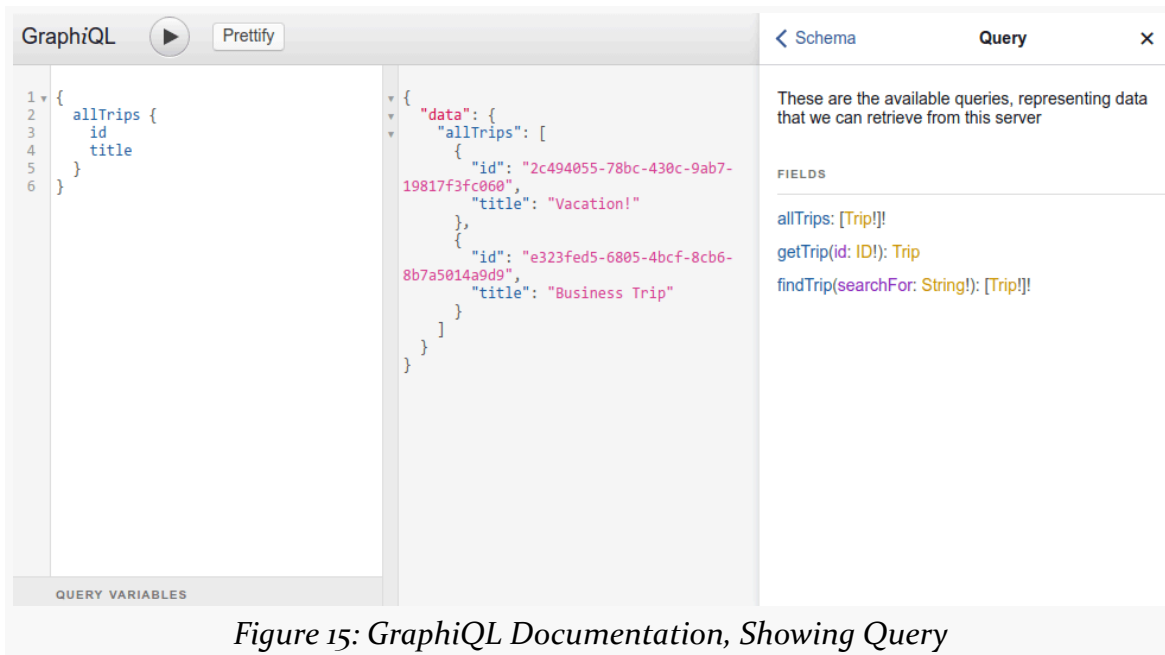
Figure 15: GraphiQL Documentation, Showing Query

Specifically, we see:

```
allTrips: [Trip!]!
```

The `Trip` is a type of object, one that is custom to our demo server. Server can define all sorts of object types, modeling the sorts of data that the server can return and/or manipulate.

For the moment, ignore the square brackets and exclamation points, as we will get into those later in this chapter.

The first page of that generated documentation, though, was a bit odd:
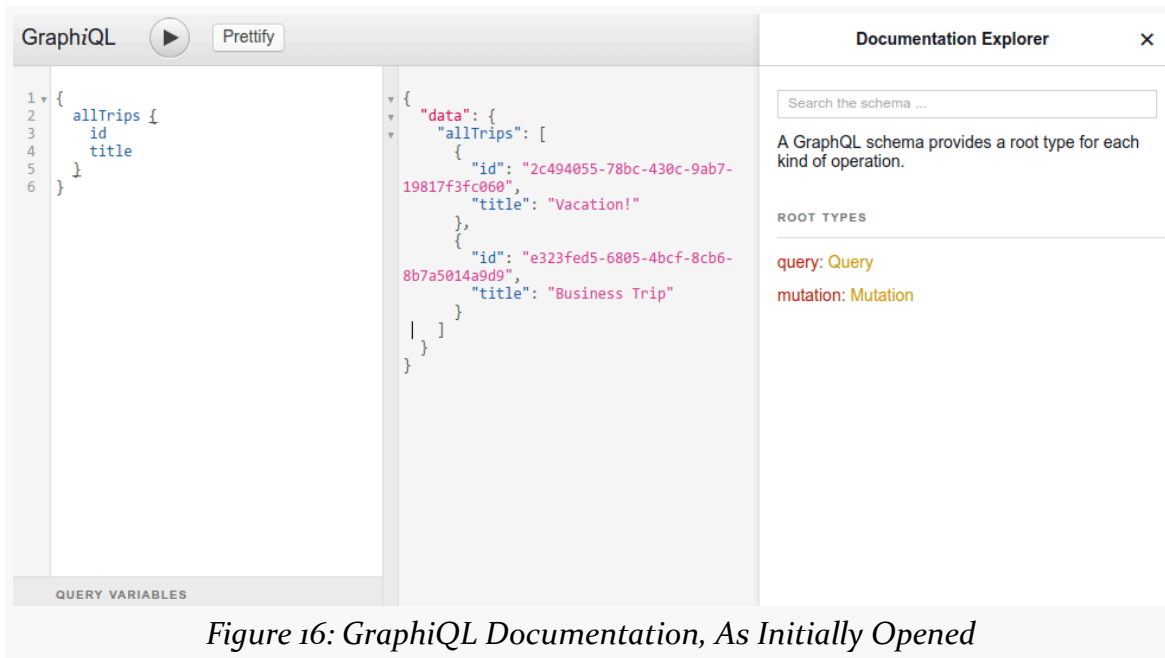
Figure 16: GraphiQL Documentation, As Initially Opened

This lists "Root Types", a `Query` named `query`, and a `Mutation` named `mutation`.

When we use the `query` or `mutation` keywords on our operation, we are telling GraphQL that we want to work with that `Query` object or that `Mutation` object. These are special objects, registered by the server, that serve as our entry points into the data that this server returns and/or manipulates.

On most pages of the generated documentation, the name that you see centered at the top is the name of the type for which you are examining documentation. So, the first screenshot shown in this chapter shows the `Query` definition, while this one shows the definition of `Trip`:
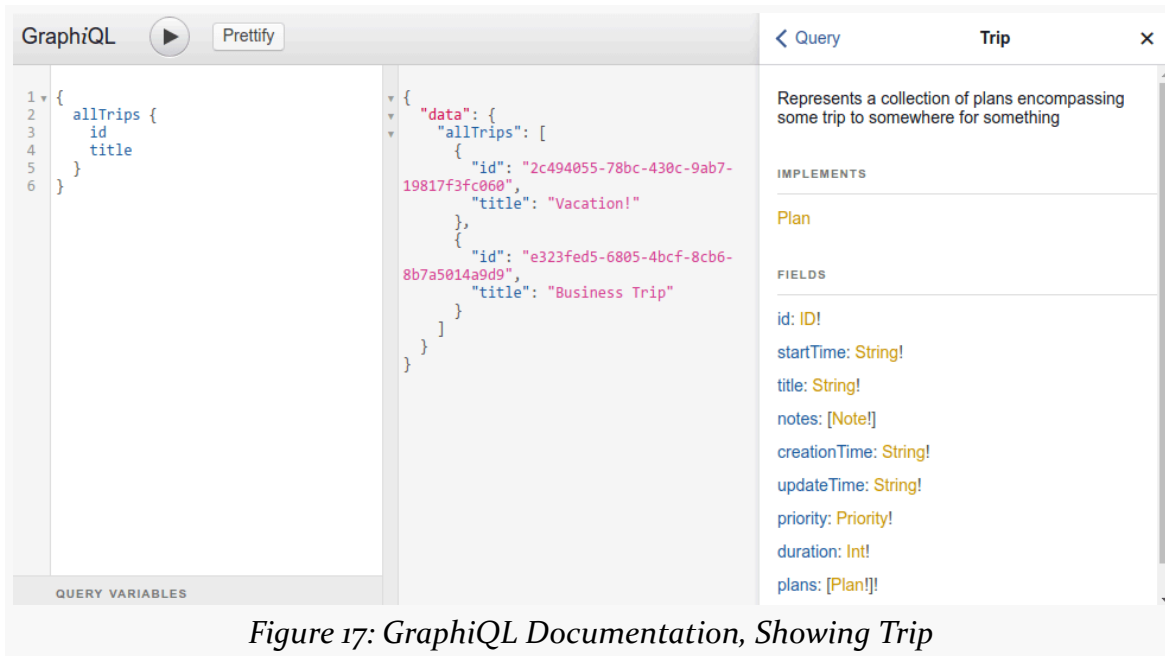
Figure 17: GraphiQL Documentation, Showing Trip

On the server, `Trip` can be defined as a custom type via the GraphQL schema definition language:

```
type Trip {

}
```

Of course, we need a bit more in that schema snippet... starting with some fields.

Fields

Objects in GraphQL have fields. When we have an operation like:

```
query all {
  allTrips {
    id
    title
  }
```

```
}
```

...what we are really saying is:

- Get the `allTrips` field from the `Query` object
- On the list of `Trip` objects that `allTrips` returns, get the `id` and `title` fields

Hence, what we retrieve via GraphQL is a nested set of fields. Roughly speaking, we describe the structure of the JSON that we want back.

If we go back to the documentation for the `Query`, we see that it has a number of fields, such as `allTrips` and `findTrips`:

Figure 18: GraphiQL Documentation, Showing Query

Some of those fields, such as `getTrip()` and `findTrips()`, look more like methods or functions, given that they have what looks like parameters inside parentheses after the name of the field. Those really are fields, but "fields" in GraphQL does not necessarily equate to "fields" as you think of them in Java or other programming languages. We will explore this more in <u>the next chapter</u>.

Root Fields

As noted earlier, a GraphQL "endpoint" (e.g., a server) will expose a `Query` object with a certain set of fields and, optionally, a `Mutation` object with its own set of

77

fields. Those are root fields, and form the top level of what we are requesting. So, `allTrips` is a root field, as it is defined on the `Query` object.

Bear in mind that the available roster of root fields may vary by circumstance, particularly via authentication and authorization:

- Anonymous users might get just some query fields, representing read access to public data
- Registered users might get some more query fields, plus some mutations, with the additional fields representing data that is unique to the user (e.g., profile settings)
- Administrative users might get the whole range of possible queries and mutations, on the grounds that administrators are local deities and can do anything

And so on. For now, we will pretend that "all users are created equal", and that the root fields are consistent.

Selection Set

The root fields are designed to return some sort of JSON structure. However, we are responsible for telling the GraphQL endpoint what fields we want out of that JSON. This forms the "selection set" of fields and drives the structure of the resulting JSON. The idea is that we can get just what we need, compared with the range of possible result data.

For example, in the query operation shown above, we are only retrieving the `id` and `title` of the trips, even though a `Trip` object has many more fields than this.

However, to understand how these selection sets work, we need to spend a little time on data types.

Data Types in GraphQL

GraphQL is strongly typed. The fields that you get back will be made up of a number of specific data types offered by GraphQL.

Numbers

Of course, we can get back numbers, both integers and floating-point values.

78

- `Int`, representing a signed 32-bit integer value
- `Float`, representing a signed double-precision value, in accordance with the IEEE 754 standard

For example, `Trip` has a `duration` field of type `Int`. So, this query:

```
{
  allTrips {
    duration
  }
}
```

...gives you JSON back with integer values for `duration`:

```
{
  "data": {
    "allTrips": [
      {
        "duration": 10080
      },
      {
        "duration": 4320
      }
    ]
  }
}
```

...given that we have `Trip` defined with a `duration` field defined as an `Int` as part of its schema:

```
  type Trip {
    duration: Int!
  }
```

We will explore that `!` on the end of `Int!` [later in this chapter](). For now, take it on faith that it means "this value cannot be `null`".

Strings and IDs

There is a `String` data type, representing text encoded in UTF-8. This is not significantly different than what you are used to in other environments. `title`, in the GraphQL shown above, is a `String`.

There is also an `ID` data type. This represents some sort of unique identifier.

GraphQL clients treat it as a string. And, it is really up to the server what the structure of an ID field is. It could be anything from an auto-incremented integer value (e.g., __ROWID__ in SQLite), a UUID, a git-style hash, etc. Clients should treat ID fields as opaque. And, ideally, GraphQL servers should be the ones to generate these ID values, so a consistent algorithm is used.

For example, Trip has an id field of type ID and a title field of type String. So, this query:

```
{
  allTrips {
    id title duration
  }
}
```

...gives you JSON back with strings for id and title:

```
{
  "data": {
    "allTrips": [
      {
        "id": "2c494055-78bc-430c-9ab7-19817f3fc060",
        "title": "Vacation!",
        "duration": 10080
      },
      {
        "id": "e323fed5-6805-4bcf-8cb6-8b7a5014a9d9",
        "title": "Business Trip",
        "duration": 4320
      }
    ]
  }
}
```

...given that we have Trip defined with those two fields:

```
type Trip {
  id: ID!
  title: String!
  duration: Int!
}
```

Custom Scalars

It is possible for a GraphQL server to define custom scalar types, beyond those listed

80

above. For example, a server might have a `DateTime` scalar type. While this will be sent back and forth in the form of a string, the actual content might be a timestamp formatted in accordance with ISO-8601 or another standard timestamp structure.

However, there is no way to know, automatically, what the structure of custom scalars are. If you are developing an app for a specific GraphQL server, it is up to the server developers to document their scalars. Otherwise, you are stuck treating them as opaque strings, much as you do with `ID` values.

GitHub's server, for example, defines a number of custom scalar types, such as:

- `DateTime`, using ISO-8601 encoding, converted into UTC
- `GitTimestamp`, which also uses ISO-8601, but is not converted into UTC
- `HTML`, representing a string containing HTML markup
- `URI`, "An RFC 3986, RFC 3987, and RFC 6570 (level 4) compliant URI string"

and so on.

For example, in GitHub's GraphiQL tool, if you execute the following GraphQL:

```
{
  viewer {
    login
    createdAt
  }
}
```

...you will get a response containing your GitHub user ID and the `DateTime` when you created your account, such as:

```
{
  "data": {
    "viewer": {
      "login": "commonsguy",
      "createdAt": "2010-08-11T22:33:53Z"
    }
  }
}
```

Enums

A GraphQL server can define a custom `Enum`, which corresponds to similar constructs in other languages, such as the enum in Java. As with a Java enum, a

GraphQL `Enum` defines a certain number of labels as the possible values. For example, a `Suit Enum` type might have `HEARTS`, `SPADES`, `DIAMONDS`, and `CLUBS` as its four options, representing the four suits in a classic Western deck of cards.

Unlike Java enum values, which typically map to integers, GraphQL `Enum` values map to the labels themselves, as strings, So, from a programming standpoint, an `Enum` is a `String` with a constrained list of possible values. So, for example, if you provide `BAMBOO` as a value for a `Suit` argument — perhaps thinking of mahjong tiles — the server will return an error.

For example, the GraphiQL-generated documentation for `Trip` shows a `priority` field that is of type `Priority`. `Priority` is an `Enum`, with four possible values:
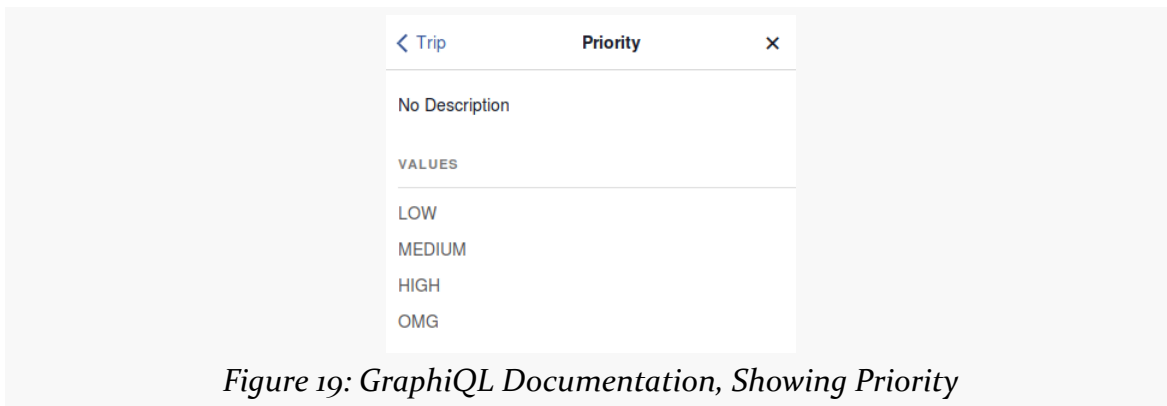
Figure 19: GraphiQL Documentation, Showing Priority

If you include `priority` in your query:

```
{
  allTrips {
    id title priority duration
  }
}
```

...you get back `Priority` values as JSON strings:

```
{
  "data": {
    "allTrips": [
      {
        "id": "2c494055-78bc-430c-9ab7-19817f3fc060",
        "title": "Vacation!",
        "priority": "MEDIUM",
        "duration": 10080
      },
```

```
    {
      "id": "e323fed5-6805-4bcf-8cb6-8b7a5014a9d9",
      "title": "Business Trip",
      "priority": "HIGH",
      "duration": 4320
    }
  ]
 }
}
```

...given that we have `Priority` defined as an enum in our schema:

```
enum Priority {
  LOW,
  MEDIUM,
  HIGH,
  OMG
}
```

...and that `Trip` has a `priority` field using that enum:

```
type Trip {
  id: ID!
  title: String!
  duration: Int!
  priority: Priority!
}
```

Objects

Many times, a field will be an object. As with a Java class or a C `struct`, an object itself contains fields. Those fields can point to any of the other data types listed in this section... including other objects. So, for example, `Trip` is an object.

Interfaces and Unions

In Java, we have interfaces. An interface stipulates a roster of methods that must be implemented by any Java class claiming to implement the interface. So, for example, the classic Android `View.OnClickListener` interface demands that any implementation have an `onClick()` method, that is `public`, returns `void`, and takes a `View` as a parameter:

```
public void onClick(View v) {
  // do something really cool
```

```
}
```

We also have abstract classes. These allow us to define part of a Java class, plus designate some methods that need to be defined in concrete subclasses. We cannot create an instance of an abstract class, just as we cannot create an instance of an interface. We can create instances of Java classes that extend abstract classes (and implement all the abstract methods), just as we can create instances of Java classes that implement interfaces.

These allow for polymorphism. We can declare method parameters, return types, and the like using interfaces and abstract classes as the data types. So, for example, `setOnClickListener()` on `View` takes a `View.OnClickListener` as a parameter:

```
btnDone.setOnClickListener(new ReallyCoolListener());
```

(where `ReallyCoolListener` must implement the `View.OnClickListener` interface)

GraphQL has two constructs that offer similar characteristics: `Interface` and `Union`.

An `Interface` is like a Java interface, except instead of requiring implementations to implement methods, they have to define certain fields.

For example, in the GraphiQL-generated documentation for `Trip`, you will see that it is marked as "Implements Plan":

Figure 20: GraphiQL Documentation, Showing Trip

`Plan` is an interface, and if you click on `Plan` in the documentation, you will bring up its definition:
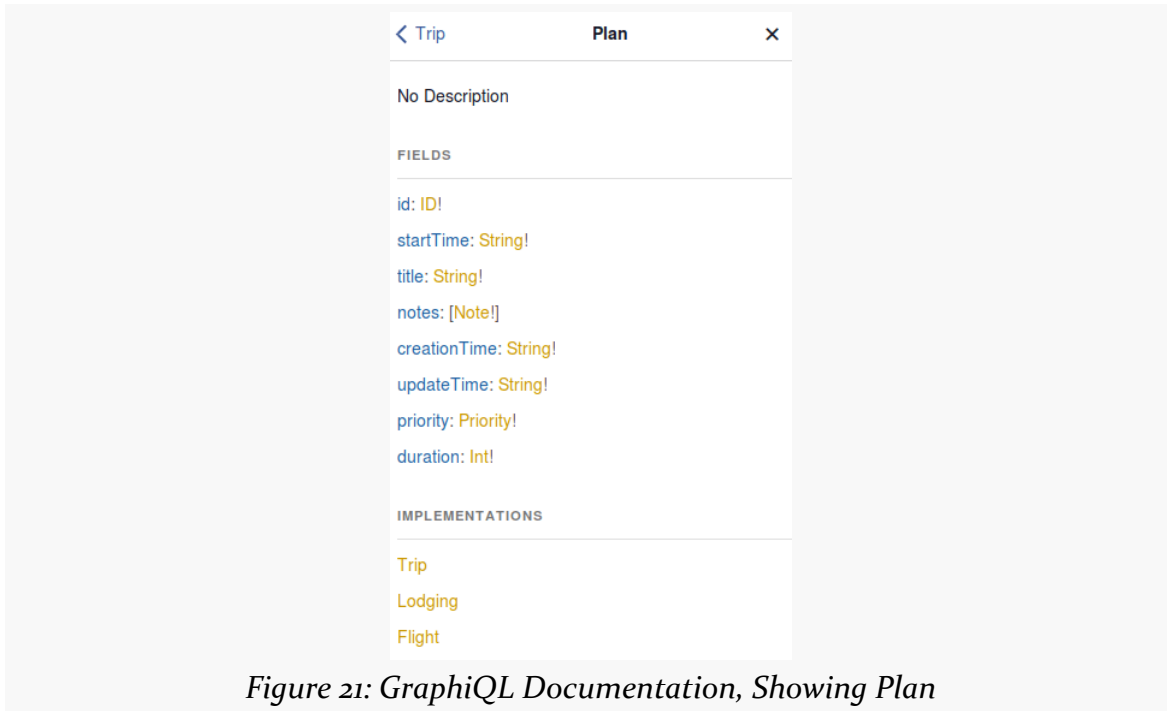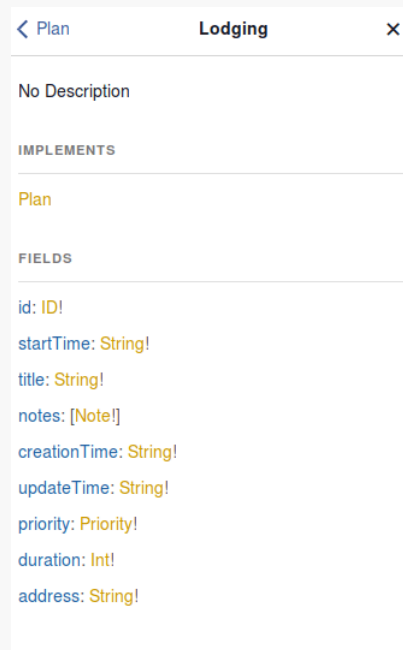
Figure 21: GraphiQL Documentation, Showing Plan

Many of the fields in `Trip` are in `Plan`. Any implementation of the `Plan` interface is obligated to have the fields from `Plan` as `Plan` defines them. The GraphiQL documentation shows other implementations of `Plan` at the bottom – not only does `Trip` implement `Plan`, but so does `Lodging` and `Flight`. So, those types also define fields from `Plan`, in addition to unique fields for their own types:

Figure 22: GraphiQL Documentation, Showing Lodging

Figure 23: GraphiQL Documentation, Showing Flight

A `Union` has no Java analogue, other than perhaps a zero-method "marker" interface like `Serializable`. A field defined as a `Union` type can be one of several possible object types… but those object types do not have to have any fields in common.

The `notes` field defined by `Plan` and implemented on `Trip`, etc. is of type `Note`. A `Note` is a `Union`, consisting of two possible types: `Comment` and `Link`:
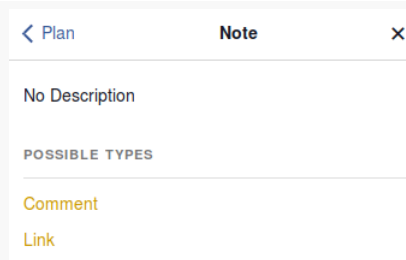
Figure 24: GraphiQL Documentation, Showing Note
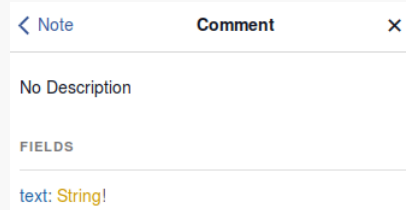
Figure 25: GraphiQL Documentation, Showing Comment
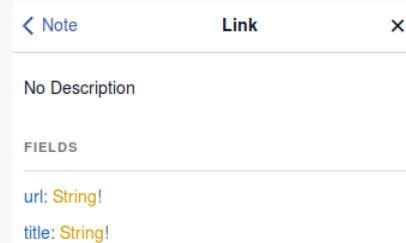
Figure 26: GraphiQL Documentation, Showing Link

`Comment` and `Link` have nothing in common. Yet, `notes` can point to either of them, as they are both part of the `Note` union.

Working with GraphQL interfaces and unions gets a bit tricky, and so we will hold off dealing with those until a later chapter.

Type Modifiers

In Java, and many other programming languages, collections are types. In Java, for example, we can have a `List` of strings, or a `HashMap` mapping IDs to customers, and so on. `List` and `HashMap` are types, not significantly different than any other Java

89

classes.

GraphQL does not work this way.

Instead GraphQL has a pair of modifiers that can be applied to types, to indicate that they are to be interpreted differently than normal. This includes a modifier for our one-and-only collection type: a list.

Lists

A type wrapped in square brackets — `[Foo]` — represents a list of that type. Here, "list" refers to what in Java we would think of as an array or `List`: an ordered sequence of values. So, whereas `Foo` represents a single value of type `Foo`, `[Foo]` represents a list of values of type `Foo`. So, for example, `allTrips` will return a list of `Trip` objects, which comes back as a JSON array of JSON objects.

Note, however, that the list could be empty. There is no requirement that a list have a certain minimum number of values. There is not documented maximum size of a list, though since this is an RPC system, your limits will be set by the time it takes to transfer large amounts of data, more so than any limits imposed by GraphQL itself.

Non-Null

A trailing exclamation point — `Foo!` — means that the value cannot be `null`. Lacking the exclamation point, we could use `null` as a valid value.

A side-effect of this: in Java, we will tend to model scalars using Java classes, not primitives, so that we can represent `null`. So, a GraphQL `Int` would be represented in Java as an `Integer`, not an `int`.

Combo Platters

Those two modifiers can be combined:

- `Foo` represents a single instance of a `Foo`, or `null`
- `Foo!` represents a single instance of a `Foo`, and cannot be `null`
- `[Foo]` represents a list of `Foo` values (or `null` values), and the entire list could be replaced by `null`
- `[Foo]!` represents a list of `Foo` values (or `null` values), but the list itself cannot be `null` (though it could be an empty list)
- `[Foo!]` represents a list of non-null `Foo` values, but the entire list could be

replaced by `null`
- `[Foo!]!` represents a list of non-`null` `Foo` values, and the list itself cannot be `null` (though it could be an empty list)

On the whole, expect `[Foo]` and `[Foo]!` to be relatively uncommon, as it is unlikely that there is much meaning to the server to have `null` values interspersed within a list.

Trip, In Schema Definition Language

`Trip`, and all of its dependent types, are baked into the `server-v0.1.js` file:

```
enum Priority {
  LOW,
  MEDIUM,
  HIGH,
  OMG
}

type Comment {
  text: String!
}

type Link {
  url: String!
  title: String!
}

union Note = Comment | Link

interface Plan {
  id: ID!
  startTime: String!
  title: String!
  notes: [Note!]
  creationTime: String!
  updateTime: String!
  priority: Priority!
  duration: Int!
}

# Represents a collection of plans encompassing some trip to somewhere for something
type Trip implements Plan {

  # A unique ID
  id: ID!

  # When this trip begins (ISO8601 date format)
  startTime: String!

  # Some human-readable identifier of this trip
  title: String!
```

```graphql
  # Text and links with additional details about this trip
  notes: [Note!]

  # Server-supplied time when this trip was created
  creationTime: String!

  # Server-supplied time when this trip was last updated
  updateTime: String!

  # How important is this trip, really?
  priority: Priority!

  # How long the trip will be, in minutes
  duration: Int!

  # The flights, lodging, etc. that make up this trip
  plans: [Plan!]!
}

type Lodging implements Plan {
  id: ID!
  startTime: String!
  title: String!
  notes: [Note!]
  creationTime: String!
  updateTime: String!
  priority: Priority!
  duration: Int!
  address: String!
}

type Flight implements Plan {
  id: ID!
  startTime: String!
  title: String!
  notes: [Note!]
  creationTime: String!
  updateTime: String!
  priority: Priority!
  duration: Int!
  departingAirport: String!
  arrivingAirport: String!
  airlineCode: String!
  flightNumber: String!
  seatNumber: String
}

input TripInput {
  startTime: String!
  title: String!
  priority: Priority!
  duration: Int!
}

# These are the available queries, representing data that  we can retrieve from this server
type Query {

  # A list of all of the trips
  allTrips: [Trip!]!
```

```
  # Obtain a trip given its unique ID
  getTrip(id: ID!): Trip

  # Find trips by searching their title and notes for a string
  findTrips(searchFor: String!): [Trip!]!
}

type Mutation {
  createTrip(trip: TripInput!): Trip!
}
```

(from [Trips/Local/server-v0.1.js](#))

At the bottom, we have definitions for the root `Query` and `Mutation` objects, with their fields. Above that, we have definitions for the types described in this chapter (e.g., `Trip`, `Plan`), along with a few things that we have not covered yet (e.g., `TripInput`).

The lines beginning with # characters are comments. Comments immediately preceding types and fields are usually treated as a description of those types and fields. Those descriptions are included as part of the [introspection API](#) that tools like GraphiQL use to generate the documentation.

Fragments

As an Android developer, you probably have experience with what the Android SDK refers to as fragments. For some of you, this has been a positive experience. For the rest of you... perhaps the experience has not been as nice.

The good news is that GraphQL fragments have nothing to do with Android fragments.

The bad news is that you will have to live with yet another overloaded use of the term "fragment".

The Role of Fragments

Oddly enough, fragments in GraphQL and Android's fragments do have a common use case: creating a reusable bit of logic. In the case of Android's fragments, we are trying to create a reusable bit of UI, to be applied in different ways in different places (e.g., smaller-screen vs. larger-screen layouts).

GraphQL fragments, on the other hand, reuse collections of fields. If there are 2+ places in our GraphQL document where we refer to the same collection of fields, we can define that collection once in a fragment, then reference the fragment where we ordinarily would have the fields themselves.

For example, suppose we have the following document:

```
query keyStuff {
  allTrips {
    id
    title
  }
```

```
}
```

That is nice and simple. Suppose we further expand the document, to:

```
query keyStuff {
  allTrips {
    id
    title
  }
}

query moreStuff {
  allTrips {
    id
    title
    priority
    duration
  }
}
```

We have some duplication, where both the `keyStuff` and `moreStuff` operations request the same `id` and `title` fields. But, two fields is not that much duplication... until we continue revising the document:

```
query keyStuff {
  allTrips {
    id
    title
    plans {
      id
      title
    }
  }
}

query moreStuff {
  allTrips {
    id
    title
    priority
    duration
    plans {
      id
      title
      priority
      duration
    }
```

```
  }
}
```

Now we have several axes of duplication:

- Both operations requests `id` and `title`
- Each operation requests `id` and `title` both for the trips and for the plans within those trips
- The `moreStuff` operation duplicates the `priority` and `duration` fields

All that duplication should seem like a code smell… because it *is* a code smell.

Fortunately, fragments can help eliminate the duplication.

Creating a Fragment

A GraphQL fragment consists of the `fragment` keyword, a name for the fragment, the data type whose fields we are interested in, and the fields themselves.

For example, we could declare a fragment for those common fields like this:

```
fragment commonFields on Plan {
  id
  title
}
```

The `on` `Plan` part will constrain where we use this fragment — we can only use it as part of querying for `Plan` objects, not something else. Plus, tools like GraphiQL can validate that our fragment is referencing actual fields from `Plan`, instead of containing a typo.

Note that `Plan` itself is an interface; `Trip` implements that interface. We will explore interfaces more later in this book. For the time being, take it on faith that `Plan` defines a bunch of common fields, and `Trip` implements those fields, as do anything in the `plans` list.

Using a Fragment

Now, we can use that fragment via a … prefix:

```
query keyStuff {
```

```
  allTrips {
    ...commonFields
    plans {
      ...commonFields
    }
  }
}

query moreStuff {
  allTrips {
    ...commonFields
    priority
    duration
    plans {
      ...commonFields
      priority
      duration
    }
  }
}

fragment commonFields on Plan {
  id
  title
}
```

We just replace our roster of fields with . . . and the name of the fragment (in this
case, commonFields). This works reminiscent of <include> elements in Android
layout resources, or #include directives in C/C++ programming: the contents of the
fragment are added, just as if we had typed them in directly.

On the whole, GraphQL is name-based, not position-based. So, even though the
fragment is defined "after" the queries that use them, this is syntactically valid. So
long as the fragment is defined in the GraphQL document, *where* in the GraphQL
document does not matter too much. This is reminiscent of Java, where so long as
methods, fields, and such are located in syntactically-valid places, their order top-
down within the Java source file does not matter.

Note that you are not limited to a single fragment. You can define as many
fragments as you need to:

```
query keyStuff {
  allTrips {
    ...commonFields
    plans {
```

```
    ...commonFields
    }
  }
}

query moreStuff {
  allTrips {
    ...commonFields
    ...extraFields
    plans {
      ...commonFields
      ...extraFields
    }
  }
}

fragment commonFields on Plan {
  id
  title
}

fragment extraFields on Plan {
  priority
  duration
}
```

Fragments, And Your Output

One key thing to note about fragments is that the output does not show any signs that you happened to use a fragment. The JSON simply reports the matched fields, such as this output from running the keyStuff operation against the demo GraphQL server:

```
{
  "data": {
    "allTrips": [
      {
        "id": "2c494055-78bc-430c-9ab7-19817f3fc060",
        "title": "Vacation!",
        "plans": [
          {
            "id": "319185bd-fab0-49e3-86ce-251d2aaa5d23",
            "title": "Flight to Chicago"
          },
          {
            "id": "319185bd-fab0-49e3-86ce-251d2aaa5d23",
```

```
          "title": "House of Munster"
        }
      ]
    },
    {
      "id": "e323fed5-6805-4bcf-8cb6-8b7a5014a9d9",
      "title": "Business Trip",
      "plans": [
        {
          "id": "d40eb2e7-3211-422e-858c-403cbe3fa680",
          "title": "Flight to Denver"
        },
        {
          "id": "e28a591b-cdc9-4328-9e79-9e4ed60ae7d2",
          "title": "Hotel Von"
        }
      ]
    }
  ]
 }
}
```

The `commonFields` fragment was referenced twice, but the label `"commonFields"` does not appear in the JSON.

Fragments and Your Android Code

Because the JSON response to a GraphQL request is unaffected by that request's use of fragments, you would expect that your Android code would be unaffected as well. And that is true… for the dynamic GraphQL pattern, just using something like OkHttp and Gson to make your request.

Apollo-Android, though, does something interesting with fragments: it wraps the fields from the fragment in their own POJO.

The [Trips/CW/StaticFragment](#) sample project is a clone of the `StaticList` project from [the chapter on Apollo-Android](#). However, this time, our GraphQL document uses a fragment:

```
query getAllTrips {
  allTrips {
    ...commonFields
    startTime
    priority
```

100

```
    duration
    creationTime
  }
}

fragment commonFields on Plan {
  id
  title
}
```

<div align="center">(from <u>Trips/CW/StaticFragment/app/src/main/graphql/com/commonsware/graphql/trips/api/TripServer.graphql</u>)</div>

In this case, using a fragment is completely pointless. However, starting in <u>the next chapter</u>, we will start to see more critical uses of fragments with Apollo-Android, so while this sample is silly, the underlying change is not.

We still get a `GetAllTrips` class from the Apollo-Android code generation, including the `GetAllTrips.Data.AllTrip` nested class. In `StaticList`, `AllTrip` directly held all the requested fields, including `title`. In theory, since the returned JSON is the same, we would get the same results despite using the fragment. In practice, that is not what Apollo-Android does.

Instead, Apollo-Android generates a `fragments()` method on `AllTrip`. `fragments()` returns an instance of a `GetAllTrips.Data.AllTrip.Fragments` class. That, in turn, has a `commonFields()` method, named after our fragment. `commonFields()` returns a `CommonFields` object, and *that* has `id()` and `title()` methods, akin to those that `StaticList` has directly on `AllTrip`.

As a result, instead of just calling `title()` on an `AllTrip` to get the title, we have to call `fragments().commonFields().title()`, just because we used a fragment. In the case of the sample app, that comes in `RowHolder`, when we try to put the title into the `RecyclerView` rows:

```
 private static class RowHolder extends RecyclerView.ViewHolder {
   private final TextView rowLabel;
   private final DateFormat dateFormat;

   RowHolder(View itemView, DateFormat dateFormat) {
     super(itemView);

     rowLabel=(TextView)itemView.findViewById(android.R.id.text1);
     this.dateFormat=dateFormat;
   }

   void bind(GetAllTrips.AllTrip trip) {
     try {
       Date parsedStartTime=ISO8601.parse(trip.startTime());
       rowLabel.setText(String.format("%s : %s",
```

<div align="center">**101**</div>

```
        dateFormat.format(parsedStartTime),
        trip.fragments().commonFields().title()));
    }
    catch (ParseException e) {
      Log.e(getClass().getSimpleName(), "Exception parsing "+trip.startTime(), e);
    }
  }
}
```

(from Trips/CW/StaticFragment/app/src/main/java/com/commonsware/graphql/trips/simple/SimpleTripsFragment.java)

Where Apollo-Android Generated Code Gets Generated

Back in [the chapter on Apollo-Android](), we saw that the GetAllTrips class was code-generated in the Java package used for the GraphQL document. That holds true for all classes that map to GraphQL operations.

However, our CommonFields class is not generated in that package. Instead, it winds up in com.commonsware.graphql.trips.api.fragment.

Similarly, other classes tied to our schema wind up in sub-packages of the schema's package. For example, Priority is an enum. Apollo-Android generates an equivalent Java enum, which goes into a .type sub-package off of the schema's Java package. So, in this project, the fully-qualified class name of Priority is com.commonsware.graphql.trips.api.type.Priority.

Arguments and Variables

The GraphQL that we have used for most of this book has been fairly simplistic. We ask for some fields, and we get those results. We have not even performed a mutation yet.

Partly, that is due to a lack of input.

The GraphQL operations that we have seen mostly have been just a tree of fields. The user provided no input for any of it.

However, we did see one operation where we provided some input:

```
query find($search: String!) {
  findTrips(searchFor: $search) {
    id title startTime priority duration
  }
}
```

Here, we are attempting to find trips based upon some supplied search criteria.

This input comes in the form of arguments (`searchFor`) and variables (`$search`). For any serious GraphQL work, you are going to use arguments and variables quite a bit. So, in this chapter, we will take a look at how these work, and along the way examine a new data type that we skipped over in the last chapter: the input object.

Arguments

When you read the word "field" in the previous chapter, you probably were thinking about fields the way normal programmers do.

103

GraphQL is not normal.

A normal programmer would think that fields are…, well, fields. They are data. So, for example, the `Trip` might be a Java POJO akin to the one that Apollo-Android code-generates for us, with `String id` and `String title` fields.

This is entirely possible with GraphQL. However, fields can also be more like methods or functions, taking arguments.

This is fairly easy to envision with root fields, as we have done that already in this book, with the `findTrips()` root field on the `Query` object:

```
query find {
  findTrips(searchFor: "ca") {
    id
    title
  }
}
```

Here, rather than retrieving all trips, we are finding trips that have `ca` somewhere in the title or notes. This makes `findTrips()` feel more like a Java method:

```
public class Query {
  List<Trip> findTrips(String searchFor) {
    // do cool stuff here to return results
  }
}
```

It is also possible to do this with nested fields, but that is a bit esoteric, so we will hold off on that topic until [later in the book](#).

Argument Data Types and Input Objects

Most of the data types described in [the preceding chapter](#) are valid data types for arguments. Specifically, all of the scalars (numbers, strings, `ID`, enums) are fine, as are lists of those. And, arguments can be marked as non-null, indicating that the argument is required.

What is *not* supported are object types (e.g., `Trip`) or things closely tied to object types, like unions (e.g., `Note`) and interfaces (e.g., `Plan`). Those can serve as output, but not input.

To help make up the gap, GraphQL has the concept of an "input object". This is

another data type, closely resembling a regular object type. However, it only supports scalars or other input objects as fields.

The GraphQL schema shown earlier had one of these, one that we just ignored at the time:

```
input TripInput {
  startTime: String!
  title: String!
  priority: Priority!
  duration: Int!
}
```

<div align="right">(from <u>Trips/Local/server-vo.1.js</u>)</div>

`TripInput` contains a few of the fields from `Trip`. However, technically, `Trip` and `TripInput` are unrelated, in terms of the schema. `TripInput` is only used in the `createTrip()` field on the `Mutation` object:

```
type Mutation {
  createTrip(trip: TripInput!): Trip!
}
```

<div align="right">(from <u>Trips/Local/server-vo.1.js</u>)</div>

It so happens that `createTrip()` takes the data from the supplied `TripInput` and creates a `Trip`. However, that is business logic implemented in the JavaScript that handles operations with `createTrip()` — there is nothing in the schema itself that says that `TripInput` has anything to do with `Trip`.

Mostly, using input objects is a way to shrink the argument list for a field. `createTrip()` could just as easily request a bunch of arguments, reflecting the fields that we happen to define on `TripInput`, rather than accepting a `TripInput`. Particularly for arguments that are logically peers of one another — such as fields that all go directly into a new `Trip` – using an input object will be a useful design pattern, but it is merely a pattern, not a requirement.

Argument Patterns

A number of common Web service API patterns can be implemented via arguments on root fields, such as those outlined in the following sections. This is not a comprehensive list of such patterns, let alone all possible ways of using arguments, but they should give you an idea of the role that arguments play.

Searching and Sorting

The searchFor argument is a crude, but simple, way of expressing a search request. The client has no means of indicating *where* we should be using that search term, or even how the search term should be used. That is up to the server, and it is up to the developers of the server to document the rules. The searchFor value could be interpreted in its own language, akin to how search engines, SQLite's FTS3/FTS4 tables, and the like use boolean algebra as part of interpreting what you supply as a search expression.

It is also possible to offer richer syntax for searching directly in the GraphQL itself. For example, we could have an input object that mirrors likely fields on Trip that allow us to provide search expressions for those specific fields (or null as a wildcard, accepting anything):

```
input TripSearchCriteria {
  title: String,
  notes: String,
  priority: Priority,
  minDuration: Int,
  maxDuration: Int
}
```

A searchTrips() field could take a TripSearchCriteria as input and apply the individual fields from those criteria (e.g., restricting results to trips with a duration in the specified range, if either or both of minDuration and maxDuration were supplied).

The expressive power available to the client is dictated by what the server wants to support. The [graph.cool CRUD store](#) offers a GraphQL interface to its database, one that code-generates a GraphQL schema [supporting expansive searching/filtering criteria](#):

```
query find {
  findTrips(
    filter: {
      OR: [
        {
          AND: [
            {
              startDate_gte: "2017"
            },
            {
```

```
            title_starts_with: "Vacation"
          }
        ]
      },
      {
        priority: "OMG"
      }
    ]
  }
) {
  id title
  }
}
```

Here, we would be retrieving any trip where the priority is `OMG` *or* the title starts with `Vacation` and is in 2017 or beyond.

Similarly, the GraphQL schema might support arguments for server-side sorting of the results (e.g., a `sortBy` argument taking the name of a field).

CRUD

With REST-style Web services, updates to resources are handled via particular HTTP actions (`PUT`, `PATCH`, etc.), with the content representing the update coming in the HTTP request body.

With GraphQL, updates to resources are handled by mutations, with the content representing the update coming in the arguments. For example, this could be in the form of an input object, so there is a single argument to the `insert()` mutation (with the content to be inserted), or two arguments to an `replace()` (with the second being the unique identifier of the content to be replaced).

We will see some CRUD [later in this chapter](#).

Paging

Many Web service APIs work off of a "page-at-a-time" metaphor, as asking for the entire data set might be too much for either the client or the server. Classic implementations include:

- Ask for pages of a certain size, and ask for a page with a certain position (e.g., "give me the 3rd 20-item page")

- Ask for N items starting with some position
- Ask for a page, where the response includes some token that allows you to get the next page

Using Arguments in Android

The `Trips/CW/SearchArgs` sample project adds a `SearchView` to the `DynamicList` sample from the chapter on dynamic GraphQL. When the user searches on something, we will use `findTrips()` to find the subset of trips that match the search expression, then show that subset in the list.

So, we have a menu resource with our `SearchView`:

```xml
<?xml version="1.0" encoding="utf-8"?>
<menu xmlns:android="http://schemas.android.com/apk/res/android">

  <item
    android:id="@+id/search"
    android:actionViewClass="android.widget.SearchView"
    android:icon="@drawable/ic_search_white_24dp"
    android:showAsAction="ifRoom|collapseActionView"
    android:title="@string/menu_search">
  </item>

</menu>
```

(from Trips/CW/SearchArgs/app/src/main/res/menu/actions.xml)

That, in turn, is set up in query mode, with the submit button enabled, in `onCreateOptionsMenu()` of our `SimpleTripsFragment`:

```java
@Override
public void onCreateOptionsMenu(Menu menu, MenuInflater inflater) {
  inflater.inflate(R.menu.actions, menu);

  search=menu.findItem(R.id.search);

  sv=(SearchView)search.getActionView();
  sv.setOnQueryTextListener(this);
  sv.setOnCloseListener(this);
  sv.setSubmitButtonEnabled(true);
  sv.setIconifiedByDefault(true);
  sv.setIconified(true);

  super.onCreateOptionsMenu(menu, inflater);
```

108

```
}
```

When the user clicks that submit button in the `SearchView`, we ask the `MainActivity` to search for the requested search expression:

```java
@Override
public boolean onQueryTextSubmit(String query) {
  ((MainActivity)getActivity()).searchFor(query);
  search.collapseActionView();

  return(true);
}
```

That, in turn, creates a new `SimpleTripsFragment`, using a new `searchFor()` factory method, and adds that new fragment to the back stack:

```java
void searchFor(String search) {
  getFragmentManager().beginTransaction()
    .replace(android.R.id.content,
      SimpleTripsFragment.searchFor(search))
    .addToBackStack(null)
    .commit();
  updateTitle(search);
}
```

`searchFor()` on `SimpleTripsFragment` just stuffs the search expression into the arguments `Bundle`:

```java
public static SimpleTripsFragment searchFor(String search) {
  SimpleTripsFragment result=new SimpleTripsFragment();
  Bundle args=new Bundle();

  args.putString(ARG_SEARCH, search);
  result.setArguments(args);

  return(result);
}
```

In `onCreate()` of `SimpleTripsFragment`, we still set up our `Observable`. However,

now, we see if we have a search expression, and we choose different sources of data based on that, calling either the original `query()` method or a new `search()` method:

```java
@Override
public void onCreate(Bundle savedInstanceState) {
  super.onCreate(savedInstanceState);

  setRetainInstance(true);
  setHasOptionsMenu(true);

  final String searchFor=getSearchExpression();

  observable=Observable
    .defer(new Callable<ObservableSource<GraphQLResponse>>() {
      @Override
      public ObservableSource<GraphQLResponse> call() throws Exception {
        return(Observable.just(searchFor==null ? query() : search(searchFor)));
      }
    })
    .subscribeOn(Schedulers.io())
    .observeOn(AndroidSchedulers.mainThread())
    .cache();
}
```

(from [Trips/CW/SearchArgs/app/src/main/java/com/commonsware/graphql/trips/simple/SimpleTripsFragment.java](#))

Using arguments without [variables](#) is... annoying, at best. We have to use string interpolation, replacing placeholders with our desired search expression, and hope for the best. That is only possible using dynamic GraphQL — code generators do not expose the GraphQL source for you to massage this way.

So, we have a `SEARCH_DOCUMENT` with our dynamic GraphQL, with a `%s` placeholder for the `searchFor` argument value:

```java
private static final String SEARCH_DOCUMENT=
  "{ findTrips(searchFor: \"%s\") { id title startTime priority duration creationTime } }";
```

(from [Trips/CW/SearchArgs/app/src/main/java/com/commonsware/graphql/trips/simple/SimpleTripsFragment.java](#))

`search()` then uses `String.format()` to add the search expression to the GraphQL document:

```java
private GraphQLResponse search(String search) throws IOException {
  HashMap<String, Object> payload=new HashMap<>();
  Gson gson=new Gson();
  String query=String.format(SEARCH_DOCUMENT, search);

  payload.put(QUERY, query);

  String body=gson.toJson(payload);
  Request request=new Request.Builder()
    .url(ENDPOINT)
```

```
    .post(RequestBody.create(MEDIA_TYPE_JSON, body))
    .build();
Response okResponse=new OkHttpClient().newCall(request).execute();

return(gson.fromJson(okResponse.body().charStream(), GraphQLResponse.class));
}
```

(from Trips/CW/SearchArgs/app/src/main/java/com/commonsware/graphql/trips/simple/SimpleTripsFragment.java)

The revised app now has a `SearchView`, initially collapsed into an icon:
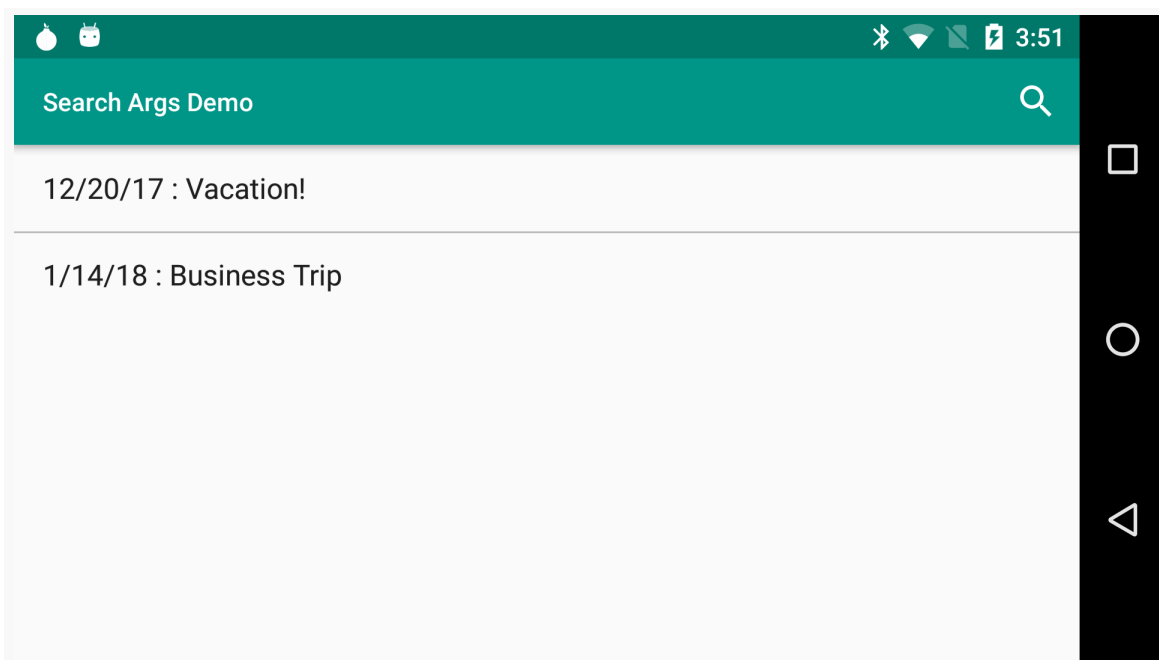
Figure 27: SearchArgs Demo App, As Initially Launched

Tapping the icon opens up the `SearchView` proper, where you can type in a search expression:
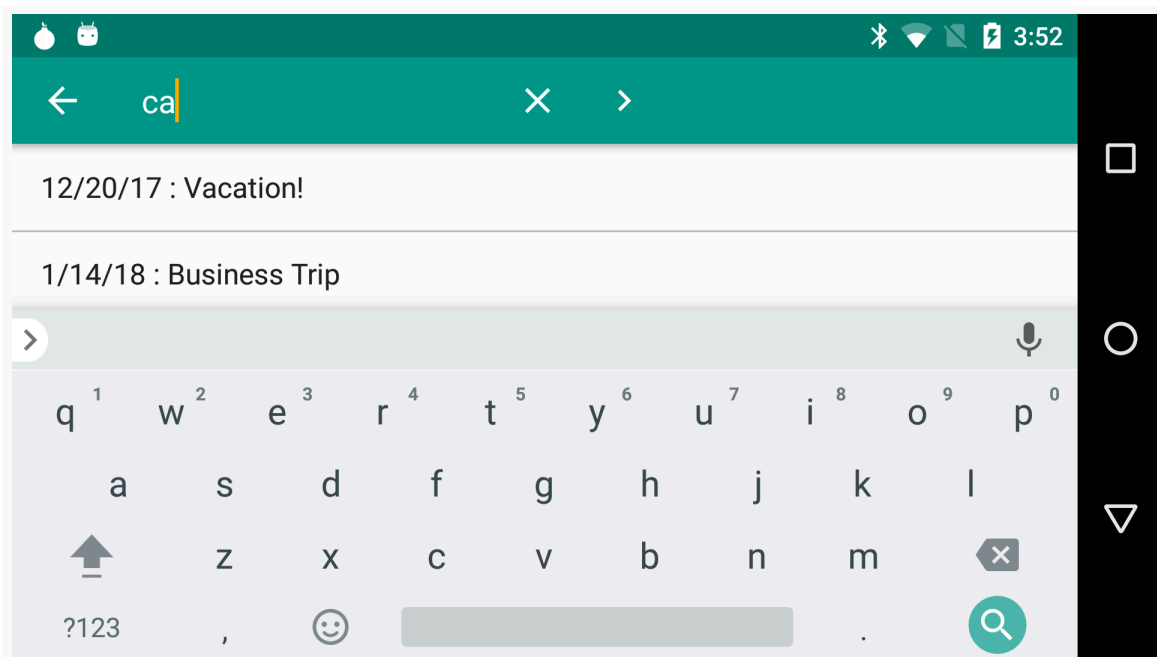
Figure 28: SearchArgs Demo App, with Search Expression

Submitting that `SearchView` (click the rightward-pointing caret) brings up a list of the search results:

Figure 29: SearchArgs Demo App, with Search Results
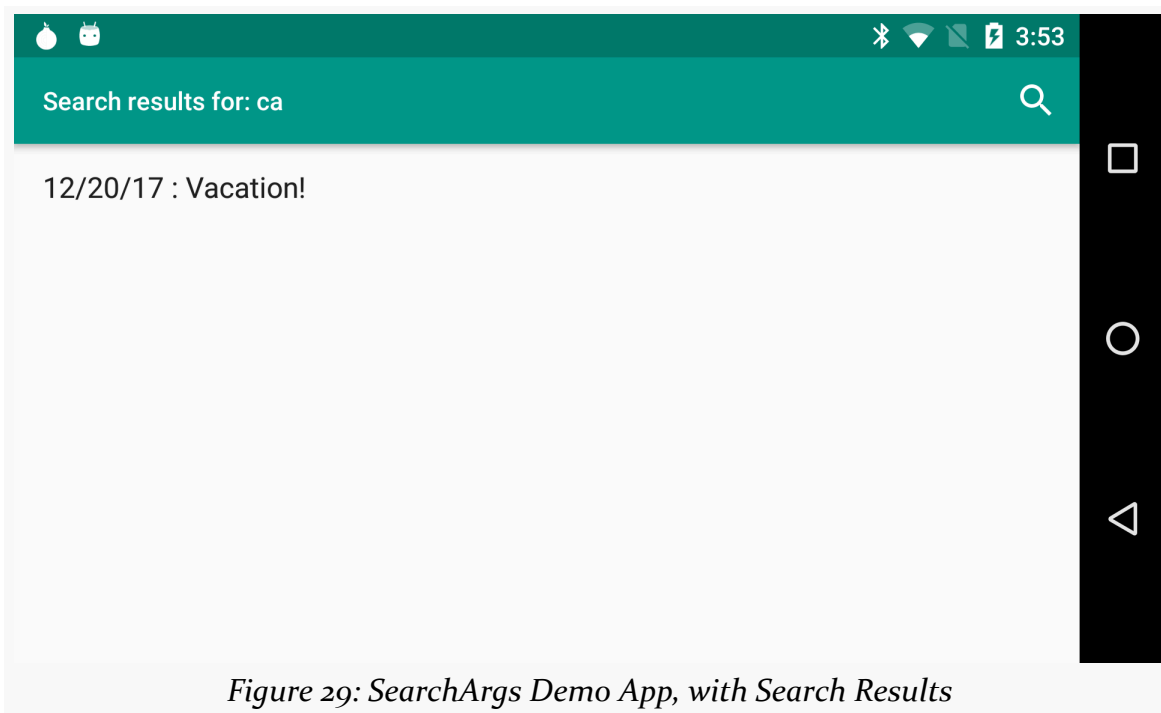
Variables

Using arguments the way that we did in that sample app, all the time, would be aggravating. Yes, GraphQL is a string, and so we can splice in our dynamic data (e.g., what the user typed into the search field). But we also have to take into account formatting rules for that data. For example, the previous sample app should fail if you try searching on something with a quotation mark, because we are not doing anything to escape that quotation mark, and we wind up with broken GraphQL. While a "Little Bobby Tables"-style attack seems unlikely, it is better not to risk it at all.

With SQLite in Android, we can use parameters:

```
SELECT * FROM foo WHERE id=?
```

Here, ? gets replaced at runtime with the first element out of a parameters array that we provide in our `rawQuery()` call against a `SQLiteDatabase`. SQLite is responsible for handling formatting (e.g., wrapping our strings in quotes and escaping anything

necessary). And, as a side-effect, we get some degree of protection against SQL injection attacks.

Similarly, we can use variables in GraphQL, to describe input to an operation, then apply that input. And, we get the same basic benefits as with SQLite parameters: automatic handling of any required escaping, and some measure of protection against what might be considered "GraphQL injection attacks".

Declaring the Variables

In a GraphQL operation, after the operation name, you can have a comma-delimited list of variable declarations:

```
query find($search: String!) {
  findTrips(searchFor: $search) {
    id
    title
    startTime
    priority
    duration
    creationTime
  }
}
```

The syntax for a variable declaration is:

- the name of the variable, of the form $ followed by some symbol
- a colon, serving as a separator
- the data type associated with the variable

So, in the above sample, we have a single variable, named `$search`, that is a non-null `String`.

Applying the Variables

Then, anywhere in your GraphQL that you have arguments, rather than hard-coding a value, you can reference the variable. In the above sample, the `searchFor` argument in `findTrips()` is now `$search` instead of `"ca"` or some other fixed string.

The data type of the argument dictates the required data type of the variable. There is no format coercion in GraphQL, the way we see in some cases in Java.

For example, suppose we have this subtly-different GraphQL operation:

```
query find($search: String) {
  findTrips(searchFor: $search) {
    id
    title
    startTime
    priority
    duration
    creationTime
  }
}
```

This will give a syntax error in GraphiQL and would result in errors if we tried asking some server to execute it. Why? Because `$search` is declared as `String`, not `String!`. Even if the value we eventually use for this variable happens to be not `null`, that does not matter.

Hence, much of the time, you will find yourself working backwards:

- Refer to the variable as the value for an argument,
- Then add the variable to the operation, using the data type of the argument where you used the variable

Supplying Values for the Variables

When we send a GraphQL request to a server, we need to send three pieces of information:

- the GraphQL document
- the GraphQL operation name, if there is more than one operation defined in the document
- a JSON object representing the variables, if the operation we chose takes variables

That JSON object will have one field per variable:

```
{
  "searchFor": "ca"
}
```

However, the value of the field is dictated by the data type of the variable. So, a field could be a string, a number, a boolean, `null`, a nested JSON object (if the variable

type is an input object), or an array of any of those things.

Servers *may* elect to do some type coercing, converting properties from the form you provided them into the form that the GraphQL schema called for. Do not assume that a server will do this. Instead, assume that the server will validate your request against the schema and return an error if you provide the wrong type (e.g., `true` instead of `"true"` for a string variable).

Typically, a complex GraphQL request is sent via an HTTP `POST` request, in which case the three pieces of the GraphQL request turn into fields of a larger JSON object that represents the request:

```
{
  "query": "query find($search: String) { findTrips(searchFor: $search) { id } }"
  "operationName": "find"
  "variables": {
    "searchFor": "ca"
  }
}
```

Variables in Android

As you might expect, the form in which you provide the variables depends a lot on what API you are using to make the GraphQL request.

Dynamic GraphQL

With dynamic GraphQL, in the end, we need the JSON representation of the entire request. But now our document can use variable declarations instead of the `%s` placeholder from before, as is seen in the [Trips/CW/SearchVarsDyn](#) sample project:

```java
 private static final String SEARCH_DOCUMENT=
   "query find($search: String!) { findTrips(searchFor: $search) { id title startTime priority duration
creationTime } }";
```

(from [Trips/CW/SearchVarsDyn/app/src/main/java/com/commonsware/graphql/trips/simple/SimpleTripsFragment.java](#))

The only significant change to our `search()` method is in replacing the `String.format()` call with the assembly of our variables:

```java
    HashMap<String, Object> payload=new HashMap<>();
    HashMap<String, String> variables=new HashMap<>();
    Gson gson=new Gson();

    variables.put(VAR_SEARCH, search);
```

116

```
    payload.put(QUERY, SEARCH_DOCUMENT);
    payload.put(VARIABLES, variables);

    String body=gson.toJson(payload);
    Request request=new Request.Builder()
      .url(ENDPOINT)
      .post(RequestBody.create(MEDIA_TYPE_JSON, body))
      .build();
    Response okResponse=new OkHttpClient().newCall(request).execute();

    return(gson.fromJson(okResponse.body().charStream(), GraphQLResponse.class));
  }
```

(from Trips/CW/SearchVarsDyn/app/src/main/java/com/commonsware/graphql/trips/simple/SimpleTripsFragment.java)

We create a separate `HashMap` for the variables, and fill in the search expression. That `HashMap` then gets added to the overall `payload` `HashMap`, for Gson to turn into the proper JSON.

Everything else works as it did with the `SearchArgs` sample app from earlier in this chapter.

Static GraphQL

The Trips/CW/SearchVarsStatic sample project does the same thing as `SearchVarsDyn`, except that it uses Apollo-Android and a static GraphQL document. You might think that this is a fairly straightforward conversion, pulling the two queries into a static GraphQL document and using the code generated by Apollo-Android. Those steps are certainly required, but the implementation gets tricky, courtesy of the way Apollo-Android generates code.

…And Never the Twain Shall Meet

The naïve implemenation of the GraphQL document might just use two queries:

```
query getAllTrips {
  allTrips {
    id
    title
    startTime
    priority
    duration
    creationTime
  }
}
```

117

```
query findTrips($search: String!) {
  findTrips(searchFor: $search) {
    id
    title
    startTime
    priority
    duration
    creationTime
  }
}
```

From the standpoint of GraphQL, this is perfectly fine, though we have some code duplication in the form of the redundant lists of fields.

However, from the standpoint of Apollo-Android, these two queries are completely unrelated… despite the fact that both return lists of `Trip` objects. In particular, our `GetAllTrips.Data.AllTrips` generated class has no relationship to `FindTrips.Data.FindTrips`.

This is a problem.

In our UI, we want to show the list of all trips and the search results using the same fragment class, the same `RecyclerView.ViewHolder`, and the same `RecyclerView.Adapter`. After all, whether we are showing all trips or some subset based on a search, we are still showing trips, and so logically the UI code should be reusable. We even request the same fields in the queries, so the subset of `Trip` data that we have is the same in either case.

Apollo-Android, like the [honey badger](#), don't care.

Since `GetAllTrips.Data.AllTrips` and `FindTrips.Data.FindTrips` are unrelated, we would need two separate `RecyclerView.ViewHolder` classes and two separate `RecyclerView.Adapter` classes, just to deal with the different Java types.

This would be not be good.

Switching to a Fragment

Fortunately, there is a workaround.

As we saw back in [the chapter on fragments](#), Apollo-Android generates a POJO-style class for each fragment that we use. More importantly, with respect to this problem,

Apollo-Android uses the *same* POJO-style class for every operation that references the fragment.

So, the `SearchVarsStatic` project introduces a `TripFields` fragment:

```graphql
query getAllTrips {
  allTrips {
    ...tripFields
  }
}

query findTrips($search: String!) {
  findTrips(searchFor: $search) {
    ...tripFields
  }
}

fragment tripFields on Trip {
  id
  title
  startTime
  priority
  duration
  creationTime
}
```

(from Trips/CW/SearchVarsStatic/app/src/main/graphql/com/commonsware/graphql/trips/api/TripServer.graphql)

Now, both `GetAllTrips.Data.AllTrips` and `FindTrips.Data.FindTrips` will get a `fragments()` method. Each of those `fragments()` methods will return a `Fragments` class that has a `tripFields()` method. And, most importantly, both `tripFields()` methods return the same `TripFields` POJO.

So, we can rewrite our `RecyclerView` logic to render the contents of the `TripFields` POJOs, and our code can remain the same regardless of which GraphQL operation we execute. There is a small amount of custom code needed to extract the `TripFields` POJOs into lists that can be used by our `RecyclerView.Adapter`, but the majority of the code will be the same between the two operations.

Updating the Observable

Because we have two operations, we need to set up our `Observable` using either a `GetAllTrips` or a `FindTrips` object:

```java
  @Override
```

119

```
public void onCreate(Bundle savedInstanceState) {
  super.onCreate(savedInstanceState);

  setRetainInstance(true);
  setHasOptionsMenu(true);

  String search=getSearchExpression();

  if (search==null) {
    observable=Rx2Apollo.from(apolloClient.query(new GetAllTrips()).watcher())
      .subscribeOn(Schedulers.io())
      .observeOn(AndroidSchedulers.mainThread())
      .map(response -> (getAllTripsFields(response)))
      .cache();
  }
  else {
    FindTrips query=FindTrips.builder().search(search).build();

    observable=Rx2Apollo.from(apolloClient.query(query).watcher())
      .subscribeOn(Schedulers.io())
      .observeOn(AndroidSchedulers.mainThread())
      .map(response -> (getFindTripsFields(response)))
      .cache();
  }
```

(from Trips/CW/SearchVarsStatic/app/src/main/java/com/commonsware/graphql/trips/simple/SimpleTripsFragment.java)

The GetAllTrips branch is the same as before. If, however, we have a search expression, we first create a FindTrips.Builder via a call to the static builder() method. That FindTrips.Builder has builder-style methods for each of the variables. In our case, there is only one, named search(), for us to supply the String search expression. We then build() the FindTrips object and use that with the ApolloClient and Rx2Apollo recipe.

Note that the map() calls for each of the two Observable objects are tied to the particular query that we are running. For GetAllTrips, we map() using getAllTripsFields(). For FindTrips, we map() using getFindTripsFields(). Both of those, however, return a List of TripFields objects, which is why we can assign both outputs to the same type of Observable:

```
    .build();
```

(from Trips/CW/SearchVarsStatic/app/src/main/java/com/commonsware/graphql/trips/simple/SimpleTripsFragment.java)

Handling the Searches

getAllTripsFields() and getFindTripsFields() work much the same, pulling the TripFields objects out of the results into a List:

```
List<TripFields> getAllTripsFields(Response<GetAllTrips.Data> response) {
```

```java
    if (response.hasErrors()) {
      throw new RuntimeException(response.errors().get(0).message());
    }

    List<TripFields> result=new ArrayList<>();

    for (GetAllTrips.AllTrip trip : response.data().allTrips()) {
      result.add(trip.fragments().tripFields());
    }

    return(result);
  }

  List<TripFields> getFindTripsFields(Response<FindTrips.Data> response) {
    if (response.hasErrors()) {
      throw new RuntimeException(response.errors().get(0).message());
    }
```

(from Trips/CW/SearchVarsStatic/app/src/main/java/com/commonsware/graphql/trips/simple/SimpleTripsFragment.java)

Our subscription remains largely the same, though now buildAdapter(), TripsAdapter, and RowHolder work with TripFields objects:

```java
    return(result);
  }

  private RecyclerView.Adapter buildAdapter(List<TripFields> trips) {
    return(new TripsAdapter(trips, getActivity().getLayoutInflater(),
      android.text.format.DateFormat.getDateFormat(getActivity())));
  }

  private static class TripsAdapter extends RecyclerView.Adapter<RowHolder> {
    private final List<TripFields> trips;
    private final LayoutInflater inflater;
    private final DateFormat dateFormat;

    private TripsAdapter(List<TripFields> trips,
                        LayoutInflater inflater, DateFormat dateFormat) {
      this.trips=trips;
      this.inflater=inflater;
      this.dateFormat=dateFormat;
    }

    @Override
    public RowHolder onCreateViewHolder(ViewGroup parent,
                                        int viewType) {
      return(new RowHolder(inflater.inflate(android.R.layout.simple_list_item_1,
        parent, false), dateFormat));
    }

    @Override
    public void onBindViewHolder(RowHolder holder,
                                 int position) {
      holder.bind(trips.get(position));
```

121

```
  }

  @Override
  public int getItemCount() {
    return(trips.size());
  }
}

private static class RowHolder extends RecyclerView.ViewHolder {
  private final TextView rowLabel;
  private final DateFormat dateFormat;

  RowHolder(View itemView, DateFormat dateFormat) {
    super(itemView);

    rowLabel=(TextView)itemView.findViewById(android.R.id.text1);
    this.dateFormat=dateFormat;
  }

  void bind(TripFields trip) {
    try {
```

(from Trips/CW/SearchVarsStatic/app/src/main/java/com/commonsware/graphql/trips/simple/SimpleTripsFragment.java)

A Little Bit of CRUD

To date, we have not done anything with mutations, and it is time that we try one of those. The [`Trips/Local/CRUD`](Trips/Local/CRUD) sample project is based on `SimpleVarsStatic`, but adds a fragment where you can define a new trip.

Running the Test Server

The fact that this project is in `Trips/Local/` versus `Trips/CW/` is not an accident. `Trips/CW/` are projects that hit the public demo GraphQL server, and that server does not support mutations. `Trips/Local/` hold projects that rely upon mutations, and so need to be run against your local test server.

Setting up the test server was covered in [a previous chapter](#). As a reminder, executing the `node server.js` from the directory containing your downloaded copy of the test server will start it up.

The test server is designed to be run on your development machine, with your Android app being used on a emulator. That being said, in principle, you can run the test server wherever, and connect to it from whatever, as you see fit. So, for example, if you want to test with Android hardware, you will need your test server to run on a computer that is reachable by that Android hardware (e.g., is on the same WiFi network or LAN segment).

Also, remember that the test server only holds its trips in memory. If you stop and restart the test server, you start over with a fresh (empty) roster of trips.

The Project and Your Test Server

The author of this book does not know where your test server is running. On the whole, that is a good thing, as it would be really creepy if by just reading this book, the book's author somehow learned about the inner workings of your office network and development machine.

However, it does mean that you have to tweak any of the `Trips/Local/` projects that you want to run.

The out-of-the-box configuration will work for a test server running on the same machine as an Android emulator. That configuration comes in the form of a `LOCAL_SERVER_URL` declaration in the `gradle.properties` file:

```
LOCAL_SERVER_URL=http://10.0.2.2:4000/0.1/graphql/
```

(from [Trips/Local/CRUD/gradle.properties](#))

This needs to be set to the base URL for your test server. For example, if you are planning on testing from Android devices, instead of the emulator, you will need to change the IP address to something that identifies your test server hardware and can be reached by those devices.

A Small Schema Reminder

The schema used by the public GraphQL demo server and your copy of the test server are very similar. However, the test server offers mutations, and the public demo server does not. That means that the downloaded `schema.json` file — used by Apollo-Android — in a `Trips/Local/` project is different than the ones from the `Trips/CW/` family of projects.

The copy of `schema.json` that accompanies the projects in the Git repo will be the correct one to use. However, just bear in mind that when you change the schema used by the server, you need to re-generate the `schema.json` file. Otherwise, the code generator will not pick up changes to that schema and may not give you the proper classes and methods.

Our New GraphQL Document

The latest edition of our GraphQL document not only has the `getAllTrips` and `findTrips` queries, but it also has a `createTrip` mutation:

```
query getAllTrips {
  allTrips {
    ...tripFields
  }
}

query findTrips($search: String!) {
  findTrips(searchFor: $search) {
    ...tripFields
  }
}

fragment tripFields on Trip {
  id
  title
  startTime
  priority
  duration
  creationTime
}

mutation createTrip($trip: TripInput!) {
  createTrip(trip: $trip) {
    id
  }
}
```

(from Trips/Local/CRUD/app/src/main/graphql/com/commonsware/graphql/trips/api/TripServer.graphql)

We could ask for whatever fields we want from the created trip. Here, we just ask for the `id`, as that is generated on the server.

What Apollo-Android Generates

As with our prior operations, Apollo-Android now generates a `CreateTrip` class that serves as our means of invoking the `createTrip` operation.

We also get a `TripInput` class, as this is the first time that we are referring to that server-defined type. As with `Priority`, `TripInput` is generated into the `com.commonsware.graphql.trips.api.type` package. Whether we have one

124

operation or several that use `TripInput`, all will refer back to this same definition.

Adding a Data Entry Fragment

The app now has a fragment for allowing the user to add a trip. The UI is defined in its own layout file:

```xml
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
  android:layout_width="match_parent"
  android:layout_height="match_parent"
  android:orientation="vertical"
  android:padding="8dp">

  <ScrollView
    android:layout_width="match_parent"
    android:layout_height="0dp"
    android:layout_weight="1">

    <LinearLayout
      android:layout_width="match_parent"
      android:layout_height="wrap_content"
      android:orientation="vertical">

      <EditText
        android:id="@+id/title"
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:hint="@string/hint_title" />

      <LinearLayout
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:gravity="center_vertical"
        android:orientation="horizontal">

        <TextView
          android:layout_width="wrap_content"
          android:layout_height="wrap_content"
          android:paddingEnd="8dp"
          android:paddingRight="8dp"
          android:text="@string/label_priority" />

        <Spinner
          android:id="@+id/priority"
          android:layout_width="wrap_content"
```

125

```xml
                    android:layout_height="wrap_content" />
        </LinearLayout>

        <LinearLayout
          android:layout_width="match_parent"
          android:layout_height="wrap_content"
          android:gravity="center_vertical"
          android:orientation="horizontal">

          <TextView
            android:id="@+id/dates"
            android:layout_width="wrap_content"
            android:layout_height="wrap_content"
            android:paddingEnd="8dp"
            android:paddingRight="8dp" />

          <ImageButton
            android:id="@+id/date_picker"
            android:layout_width="wrap_content"
            android:layout_height="wrap_content"
            android:contentDescription="@string/label_pick_trip_date_range"
            android:src="@drawable/ic_event_white_24dp" />
        </LinearLayout>
      </LinearLayout>
    </ScrollView>

    <LinearLayout
      style="?android:attr/buttonBarStyle"
      android:layout_width="match_parent"
      android:layout_height="wrap_content">

      <Button
        android:id="@+id/save"
        style="?android:attr/buttonBarButtonStyle"
        android:layout_width="0dp"
        android:layout_height="wrap_content"
        android:layout_weight="1"
        android:text="@string/label_save" />

      <Button
        android:id="@+id/cancel"
        style="?android:attr/buttonBarButtonStyle"
        android:layout_width="0dp"
        android:layout_height="wrap_content"
        android:layout_weight="1"
        android:text="@string/label_cancel" />
    </LinearLayout>
</LinearLayout>
```

We have an `EditText` for the title, a `Spinner` to select the priority, and an `ImageButton` that will be used to pick the start and end dates for the trip (which, in turn, determine the trip duration). Those are wrapped in a `ScrollView`, so that when the input method editor appears for the user to type in the title, that the bottom bar, with the Save and Cancel buttons, remains in view.

The fragment — `AddTripFragment` — inflates that layout and ties the `ImageButton` to a third-party date-range dialog named `DatePickerDialog`. The Save button is tied to a `save()` method, where we will invoke the mutation.

Invoking a Mutation

`save()` needs to fill in a `TripInput` object, as that is how we supply the details of our new trip to the server:

```java
private void save() {
  EditText title=(EditText)getView().findViewById(R.id.title);
  int duration=(int)((end.getTimeInMillis()-start.getTimeInMillis())/60000);

  if (duration<0) {
    duration=-1*duration;
  }

  TripInput trip=TripInput.builder()
    .priority(PRIORITIES[priority.getSelectedItemPosition()])
    .title(title.getText().toString())
    .startTime(MainActivity.ISO8601.format(start.getTime()))
    .duration(duration)
    .build();
  final CreateTrip vars=CreateTrip.builder().trip(trip).build();

  new Thread() {
    @Override
    public void run() {
      try {
        Response<CreateTrip.Data> response=getApolloClient().mutate(vars).execute();
        EventBus.getDefault().post(new TripCreatedEvent(response));
      }
      catch (ApolloException e) {
        EventBus.getDefault().post(new TripCreatedEvent(e));
      }
    }
  }.start();
}
```

Apollo-Android generates a `TripInput.Builder` class, and we get an instance of that via `TripInput.builder()`. Then, we call methods on the builder to fill in:

- the priority, based on the selected `Spinner` position and an array of `Priority` objects:

```
private static final Priority[] PRIORITIES={
  Priority.LOW,
  Priority.MEDIUM,
  Priority.HIGH,
  Priority.OMG
};
```

(from Trips/Local/CRUD/app/src/main/java/com/commonsware/graphql/trips/crud/AddTripFragment.java)

- the title from the `EditText`
- the start time, stored in a `Calendar` field, formatted into ISO8601 format (the format the server wants the dates in)
- the duration, calculated based on the difference between the start and end dates

Then, we `build()` the `TripInput` itself.

That needs to be a variable on our `createTrip` mutation. So, we create a `CreateTrip.Builder` (via `CreateTrip.builder()`), supply it the `trip`, and `build()` the `CreateTrip`.

In principle, you could use RxJava more or less as was done elsewhere. However, bear in mind that nothing about Apollo-Android — let alone GraphQL in general – requires the use of RxJava. To demonstrate this, we just fork an ordinary `Thread`. In there, we still call `newCall()` on an `ApolloClient`, though this time the `ApolloClient` is defined in `MainActivity`, so it can be shared between the two fragments (`SimpleTripsFragment` and `AddTripFragment`). And, rather than setting up an `Observable`, we `execute()` the `Call` returned by `newCall()`, much as how you `execute()` an OkHttp request. This returns our `Respnose` of `CreateTrip.Data` synchronously. That gets wrapped in a `TripCreatedEvent`, whether we succeed (and have a `Response`) or catch an `Exception`:

```
package com.commonsware.graphql.trips.crud;

import com.apollographql.apollo.api.Response;
import com.commonsware.graphql.trips.api.CreateTrip;

class TripCreatedEvent {
  final Response<CreateTrip.Data> response;
  final Exception exception;
```

```
  TripCreatedEvent(Response<CreateTrip.Data> response) {
    this.response=response;
    this.exception=null;
  }

  TripCreatedEvent(Exception e) {
    this.exception=e;
    this.response=null;
  }
}
```

(from Trips/Local/CRUD/app/src/main/java/com/commonsware/graphql/trips/crud/TripCreatedEvent.java)

Then, that `TripCreatedEvent` is posted on an event bus, supplied by greenrobot's EventBus.

`AddTripsFragment` subscribes on the bus and picks up its own event, on the main application thread, to show a `Toast` based on the outcome and remove this fragment if we are done with it:

```
@Subscribe(threadMode =ThreadMode.MAIN)
public void onTripCreated(TripCreatedEvent event) {
  Response<CreateTrip.Data> r=event.response;

  if (event.exception!=null) {
    Toast
      .makeText(getActivity(), event.exception.getMessage(),
        Toast.LENGTH_LONG)
      .show();
    Log.e(getClass().getSimpleName(), "Exception creating trip", event.exception);
  }
  else if (r.errors()!=null && r.errors().size()>0) {
    Toast
      .makeText(getActivity(), r.errors().get(0).message(),
        Toast.LENGTH_LONG)
      .show();
  }
  else {
    Toast
      .makeText(getActivity(), R.string.msg_created, Toast.LENGTH_SHORT)
      .show();
    getFragmentManager().popBackStack();
  }
}
```

(from Trips/Local/CRUD/app/src/main/java/com/commonsware/graphql/trips/crud/AddTripFragment.java)

And, `SimpleTripsFragment` also subscribes on the bus to pick up this event, so that it knows that its list is out of date, so it should re-query the server and get all the trips.

The Results

By default, the test server has no trips:
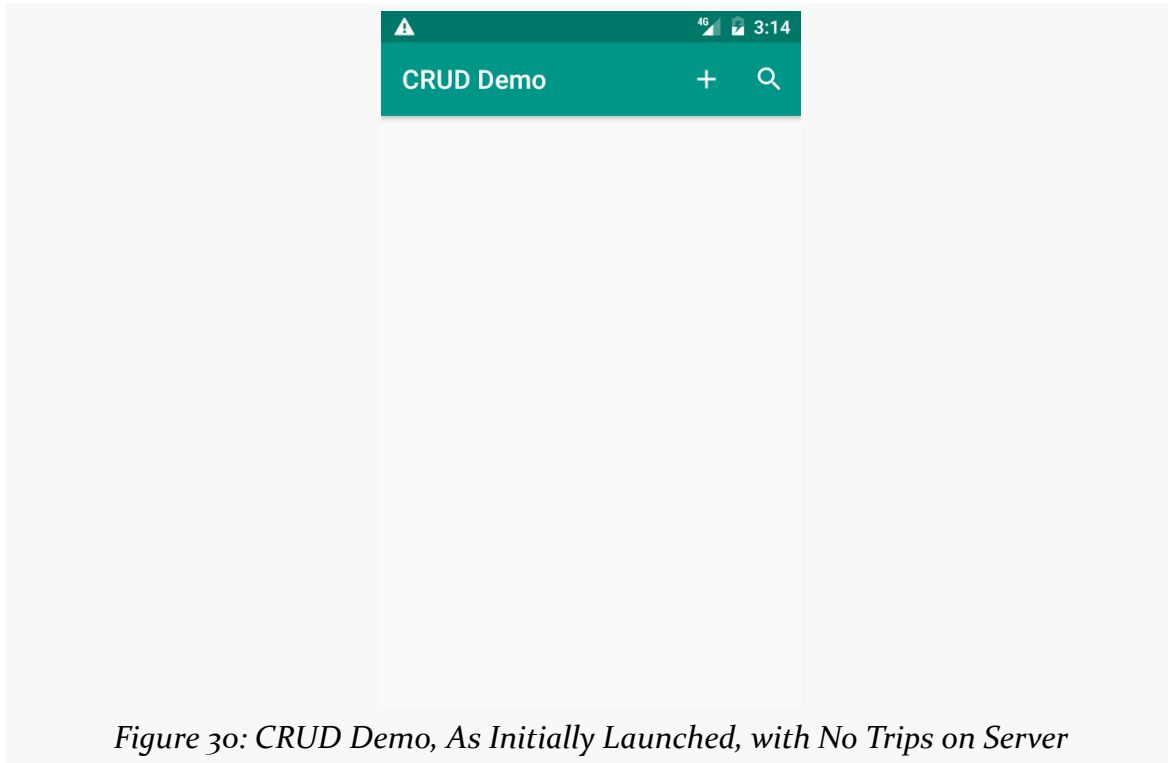
Figure 30: CRUD Demo, As Initially Launched, with No Trips on Server

The `MainActivity` now has an action bar item, shaped like a plus sign (+), to add a new trip. Tapping that brings up the UI from the `AddTripFragment`:

Figure 31: AddTripFragment, As Initially Launched
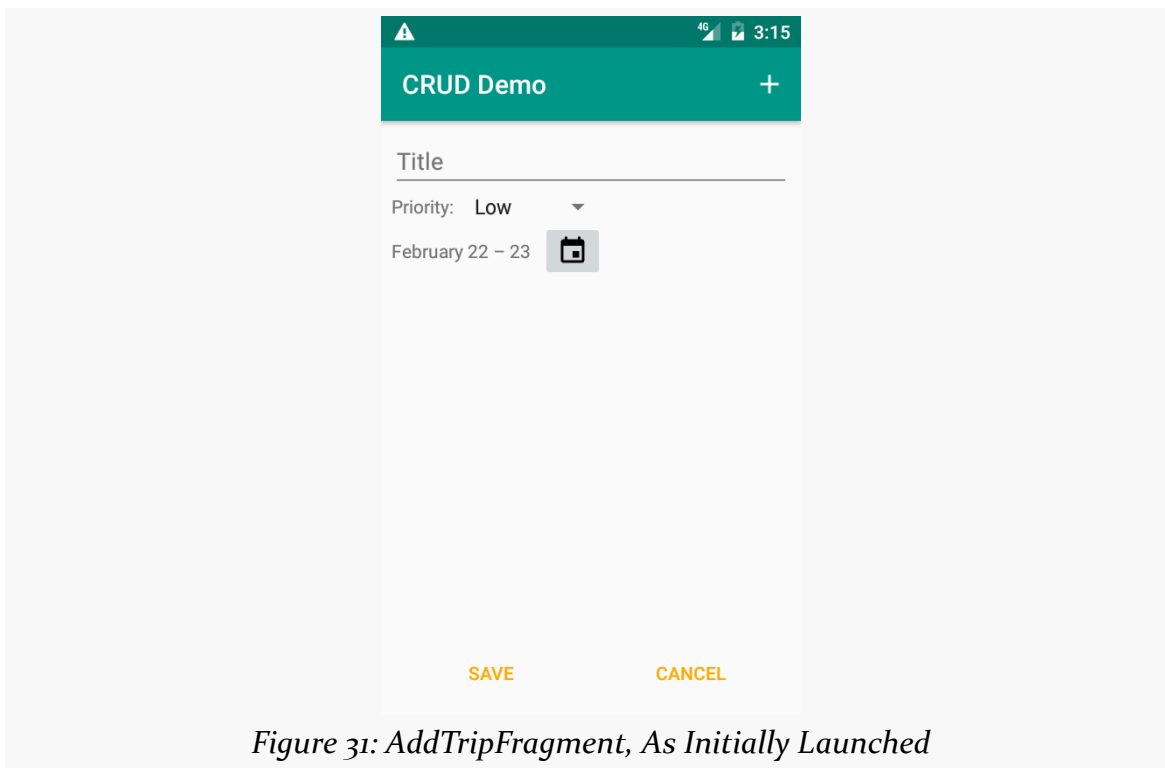
Tapping the `ImageButton` brings up the date picker, with two tabs for setting the start and end dates:
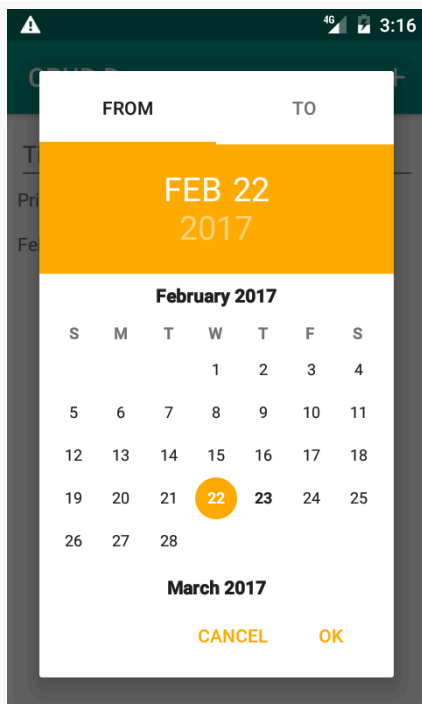
Figure 32: DatePickerDialog

Filling in the form, then clicking Save, will show your new trip in the list:
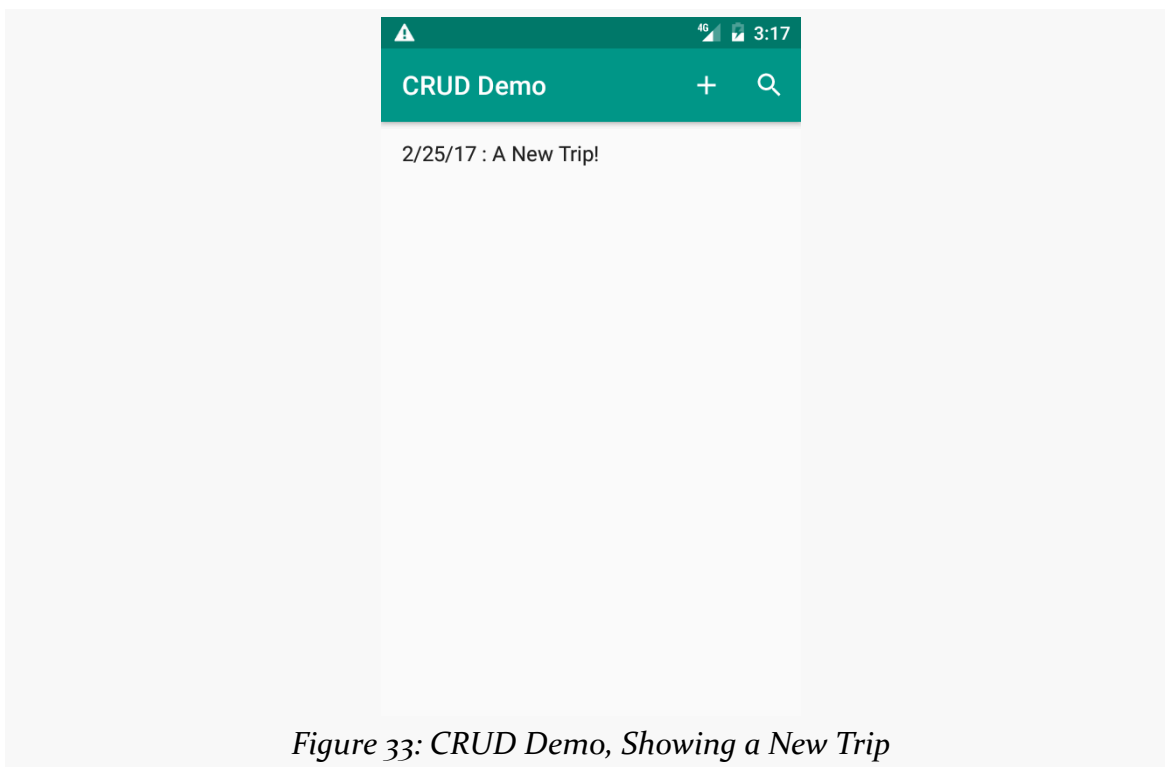
Figure 33: CRUD Demo, Showing a New Trip

And, if you bring up GraphiQL on your test server (e.g., `http://127.0.0.3/0.3/ graphql`, replacing that IP address with one that reaches your test server) and run the `allTrips` query, you will see that the new trip did indeed make it to the server:

Aliases

Phil Karlton [coined a popular programming maxim](#):

> There are only two hard things in Computer Science: cache invalidation and naming things.

Naming things is a challenge because not everbody agrees on the name. Sometimes, we cannot even agree on a naming convention (such as a `naming_convention`, a `namingConvention`, or a `NamingConvention`).

Aliases allow you, on the client side, to tell the server how to name things. These aliases then impact the JSON that is returned to you.

On the surface, this may not seem all that interesting, as developers are used to having to conform to somebody else's `NAMING_CONVENTION`. However, due to the way that GraphQL allows clients to "shape" the responses coming from the server, aliases also play a role in batched requests.

Applying Aliases

Let's go back to one of the earlier GraphQL requests that we made:

```
query all {
  allTrips {
    id
    title
  }
}
```

That gives a raw JSON result that looks like:

135

```
{
  "data": {
    "allTrips": [
      {
        "id": "2c494055-78bc-430c-9ab7-19817f3fc060",
        "title": "Vacation!"
      },
      {
        "id": "e323fed5-6805-4bcf-8cb6-8b7a5014a9d9",
        "title": "Business Trip"
      }
    ]
  }
}
```

The data gives us an allTrips property, containing the data that we requested — in this case, an array of Trip objects, for which we only wanted the id and title fields.

The reason that property is named allTrips is because we requested the allTrips field from the root query object. The field that you query (or mutate) is the default name used for the results from that query or mutation.

However, through aliases, we can rename that property to something else:

```
query all {
  stuff: allTrips {
    id
    title
  }
}
```

Here, we have applied an alias of stuff to the allTrips field. Now, our response has a stuff property instead of an allTrips property:

```
{
  "data": {
    "stuff": [
      {
        "id": "2c494055-78bc-430c-9ab7-19817f3fc060",
        "title": "Vacation!"
      },
      {
        "id": "e323fed5-6805-4bcf-8cb6-8b7a5014a9d9",
        "title": "Business Trip"
      }
    ]
```

```
  }
}
```

This may not seem especially useful. However, aliases are key to unlocking a few other GraphQL features, particularly for cases where we cannot use the original field names... perhaps because we are querying or mutating that field more than once.

One, Two, Many, Lots

(the title of this section comes courtesy of the late Terry Pratchett and the Discworld novels)

So far, we have requested just one field in our queries, and mutated just one field in our mutations. You might think that this is a limitation of GraphQL.

It's not.

It is just a limitation of trying to teach GraphQL, slowly advancing through its roster of features. In truth, a single query can retrieve many fields, and a single mutation can modify lots of fields.

Multiple Root Fields

The queries that we have used so far have requested just one root field. You can request as many as you want:

```
query allAndFind($search:String!) {
  allTrips {
    id
    title
  }
  findTrips(searchFor:$search) {
    id
    title
  }
}
```

Executing the allAndFind operation will return results for both the allTrips field and the findTrips field (given a value for $search):

```
{
  "data": {
    "allTrips": [
```

137

```
      {
        "id": "2c494055-78bc-430c-9ab7-19817f3fc060",
        "title": "Vacation!"
      },
      {
        "id": "e323fed5-6805-4bcf-8cb6-8b7a5014a9d9",
        "title": "Business Trip"
      }
    ],
    "findTrips": [
      {
        "id": "2c494055-78bc-430c-9ab7-19817f3fc060",
        "title": "Vacation!"
      }
    ]
  }
}
```

As you can see in the results, each field's results gets its own field in the JSON output, keyed by the GraphQL field's name.

Making the Same Query Over and Over and Over

Since the JSON output is organized by field name, we cannot query the same field twice by default:

```
query allAndFind($search:String!,$search2:String!) {
  findTrips(searchFor:$search) {
    id
    title
  }
  findTrips(searchFor:$search2) {
    id
    title
  }
}
```

If you try this, you will get an error back:

```
{
  "errors": [
    {
      "message": "Fields \"findTrips\" conflict because they have differing
arguments. Use different aliases on the fields to fetch both if this was
intentional.",
      "locations": [
```

```
      {
        "line": 2,
        "column": 3
      },
      {
        "line": 6,
        "column": 3
      }
    ]
  }
]
}
```

The error suggests the solution: aliases. By adding unique aliases to the fields, you provide unique keys for the JSON output, and GraphQL is happy:

```
query allAndFind($search:String!,$search2:String!) {
  find: findTrips(searchFor:$search) {
    id
    title
  }
  find2: findTrips(searchFor:$search2) {
    id
    title
  }
}
```

Given variables like:

```
{
  "search": "ca",
  "search2": "foo"
}
```

...we get results like:

```
{
  "data": {
    "find": [
      {
        "id": "2c494055-78bc-430c-9ab7-19817f3fc060",
        "title": "Vacation!"
      }
    ],
    "find2": []
  }
```

```
}
```

Mutating N Items

You can use the same approach to perform multiple mutations in a single operation. As with multiple queries, you are welcome to mutate one or more fields in an operation. If you wish to mutate the same field — such as inserting multiple objects — you can use aliases to have each result be identified separately in the GraphQL response.

For example, you could have this GraphQL operation:

```
mutation addTwo($trip1: TripInput!, $trip2: TripInput!) {
  tripOne: createTrip(trip: $trip1) {
    id
  }
  tripTwo: createTrip(trip: $trip2) {
    id
  }
}
```

Here, we insert two trips, using two `TripInput` objects:

```
{
  "trip1": {
    "title": "Foo",
    "priority": "OMG",
    "startTime": "2018-05-10",
    "duration": 10000
  },
  "trip2": {
    "title": "Bar",
    "priority": "MEDIUM",
    "startTime": "2018-07-11",
    "duration": 24000
  }
}
```

We then get back the IDs of the two newly-created trips:

```
{
  "data": {
    "tripOne": {
      "id": "62f92a4e-96f5-4207-b5c9-885e0cb4f3ad"
    },
```

```
    "tripTwo": {
      "id": "95d50515-20cd-45c2-ad5f-423ca756c9c3"
    }
  }
}
```

Aliases with Apollo-Android

Using aliases in dynamic GraphQL is "no big deal". You just need to remember to look for the alias, not the server-defined name, when parsing the results.

Apollo-Android does what you might expect: renames everything based on your aliases, replacing whatever their original names had been.

The [Trips/CW/StaticAlias](#) sample project is cloned from the SearchVarsStatic sample, but changes the GraphQL document to use aliases in three places:

```
query getAllTrips {
  result: allTrips {
    ...tripFields
  }
}

query findTrips($search: String!) {
  result: findTrips(searchFor: $search) {
    ...tripFields
  }
}

fragment tripFields on Trip {
  id
  displayName: title
  startTime
  priority
  duration
  creationTime
}
```

(from [Trips/CW/StaticAlias/app/src/main/graphql/com/commonsware/graphql/trips/api/TripServer.graphql](#))

First, both queries have their root fields renamed to result. In addition, the title field in the tripFields fragment is given an alias of displayName.

That triggers changes to the generated code:

141

- The `GetAllTrips.Data.AllTrips` class is now `GetAllTrips.Data.Result`
- The `FindTrips.Data.FindTrips` class is now `FindTrips.Data.Result`
- The `allTrips()` method is now `result()`, on both `GetAllTrips.Data` and `FindTrips.Data`
- The `title()` method on `TripFields` is now `displayName()`

GraphQL Execution Rules

Now that we are getting into potentially having multiple root fields in a single operation, the question comes up: how does the server process those fields?

According to [the GraphQL specification](#):

- A query operation with multiple root fields can do whatever it wants, including executing them in parallel
- A mutation operation with multiple root fields must execute those in serial order

Mostly, this matters for GraphQL server developers, making sure that all query fields are "side effect-free and idempotent" (to quote the specification), and making sure that they execute each mutation's fields in the order specified.

However, one key thing to note with employing multiple fields is that, as with any Web service, there is nothing about GraphQL that implies transactional integrity. For example, let's go back to the multiple-mutation operation:

```
mutation addTwo($trip1: TripInput!, $trip2: TripInput!) {
  tripOne: createTrip(trip: $trip1) {
    id
  }
  tripTwo: createTrip(trip: $trip2) {
    id
  }
}
```

Here, there are four possible outcomes:

- Both `createTrip()` invocations succeed, which is the desired outcome
- `tripOne` fails
- `tripTwo` fails
- Both fail

If there is a failure, the `errors` that come back with the `data`, coupled with existing or missing properties of the operation results in the `data`, will give you some idea of what worked and what did not.

However, there is no way for you to specify that the middle two outcomes (one succeeds and one fails) are unacceptable. In other words, there is no way for you to wrap these `createTrip()` invocations in a transaction, so that either both succeed or both fail.

Worse, due to some quirks in the GraphQL execution rules, a server cannot really *offer* a transactional guarantee, at least while maintaining fidelity with the GraphQL specification. The specification states that the server must complete each root field's processing before continuing to the next one. Or, as the specification phrases it:

> It must determine the corresponding entry in the result map for each item to completion before it continues on to the next item in the grouped field set

Partly, this is because GraphQL is not necessarily going to be used to provide a Web service API to something that offers some sort of two-phase commit transaction logic. A mutation operation that requests N distinct root fields might not just be hitting some single database, for which the server could conceivably use a transaction to wrap all those requests. Instead the server might distribute those root field requests to N distinct microservices, with no ability to have all of those microservices "succeed or fail as a whole", to use the typical phrase associated with transactions.

Any error in processing a field should result in an entry being added to the `errors` list that comes back in the GraphQL response. In theory, one can use that information to determine if `tripOne`, `tripTwo`, or both failed. In practice, since the error information is not rigorously defined, you will need to have some conventions between client and server developers as to how to determine, based on reported errors, what succeeded and what failed.

Interfaces, Unions, and Inline Fragments

Earlier in the book, we introduced the concept of [interfaces and unions](). To recap:

- An interface represents a collection of fields; types implementing that interface must contain all those fields, and fields of an interface type can hold any of the implementations of that interface
- A union represents a logical OR of multiple types; fields of a union type can hold any of the types identified by the union

The problem in both cases is that we do not know, at the time of executing a query, what we are going to get back, in terms of types. Both interface and union fields can hold any candidate type, and so our results might well have a mix of different types in the results. However, not all of the fields of those types are necessarily in common:

- Interface implementations have *some* fields in common (those defined on the interface) but can have other fields that are distinct
- Types that are part of a union need not have *any* fields in common

So... if we are trying to get data back from interface or union fields, and we do not know when we are writing the query what we are going to get back, how do we ask for those fields' contents?

The answer lies in what are known as "inline fragments". These effectively amount to a Java `switch` statement, identifying field sets to return for an interface or union type, based on the actual type for a given field.

So, let's dive into how we actually query on interfaces and unions, so we can see

145

inline fragments in action.

Interfaces

As was noted earlier in the book, in terms of our "trips" sample, we have an interface named `Plan`:
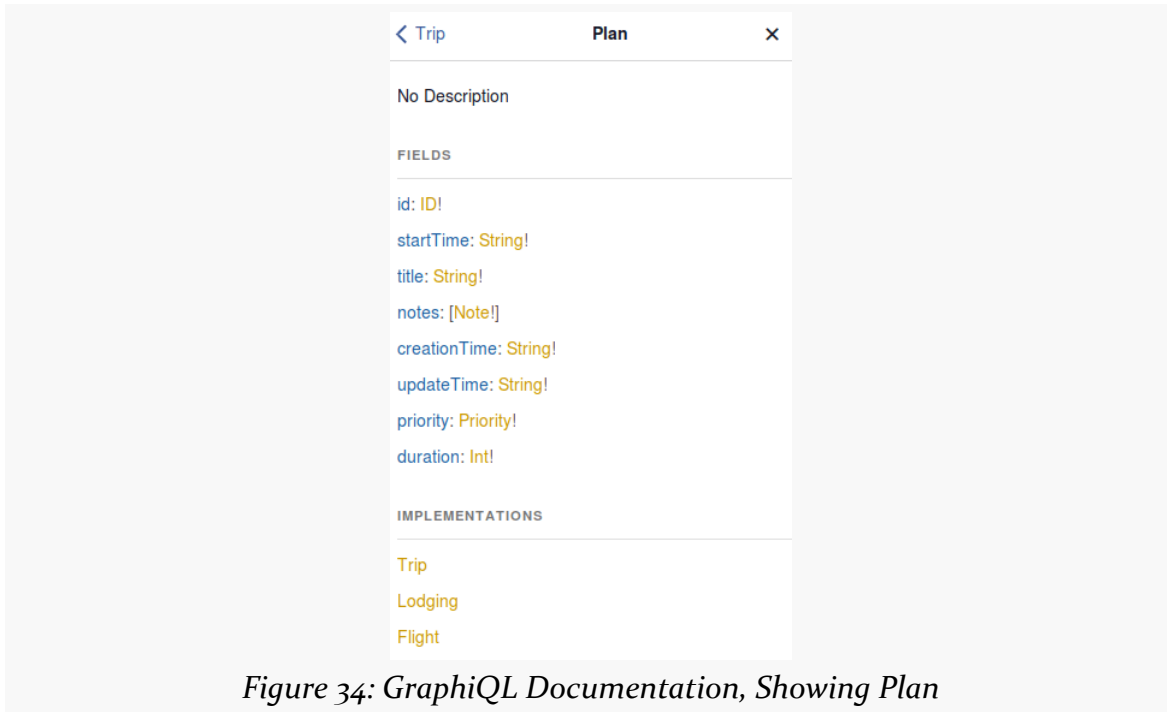
Figure 34: GraphiQL Documentation, Showing Plan

There are three types that implement `Plan`: `Trip`, `Lodging` (representing something like a hotel stay), and `Flight`. And, a `Trip` has a `plans` field that is a list of `Plan` objects, representing the individual elements in that trip's itinerary. Those objects could be `Lodging` objects, `Flight` objects, or even other `Trip` objects (e.g., for modeling a complex trip as a series of sub-trips).

So, if we want to get the `plans` for a `Trip`, how do we do that?

Fields In Common

We can easily query for the fields that are in the `Plan` interface, as all three `Plan` implementations will have those fields. We did this, in effect, back in the chapter on fragments. There, we defined a `tripFields` fragment, saying that the fields were

from `Trip`. In reality, all those fields are defined on `Plan`. So, we can revise the fragment to reference `Plan` instead of `Trip`, and now we can use that fragment for querying a trip's plans:

```
query getAllTrips {
  allTrips {
    ...planFields
    plans {
      ...planFields
    }
  }
}

fragment planFields on Plan {
  id
  title
  startTime
  priority
  duration
  creationTime
}
```

Running this against the public GraphQL demo server will give you something like:

```
{
  "data": {
    "allTrips": [
      {
        "id": "2c494055-78bc-430c-9ab7-19817f3fc060",
        "title": "Vacation!",
        "startTime": "2017-12-20T13:14:00-05:00",
        "priority": "MEDIUM",
        "duration": 10080,
        "creationTime": "2017-02-19T15:21:58.547Z",
        "plans": [
          {
            "id": "319185bd-fab0-49e3-86ce-251d2aaa5d23",
            "title": "Flight to Chicago",
            "startTime": "2017-12-20T13:14:00-05:00",
            "priority": "HIGH",
            "duration": 150,
            "creationTime": "2017-02-19T15:21:58.547Z"
          },
          {
            "id": "319185bd-fab0-49e3-86ce-251d2aaa5d23",
            "title": "House of Munster",
            "startTime": "2017-12-20T15:00:00-05:00",
```

```
          "priority": "MEDIUM",
          "duration": 9900,
          "creationTime": "2017-02-19T15:21:58.547Z"
        }
      ]
    },
    {
      "id": "e323fed5-6805-4bcf-8cb6-8b7a5014a9d9",
      "title": "Business Trip",
      "startTime": "2018-01-14T11:45:00-05:00",
      "priority": "HIGH",
      "duration": 4320,
      "creationTime": "2017-02-19T15:21:58.547Z",
      "plans": [
        {
          "id": "d40eb2e7-3211-422e-858c-403cbe3fa680",
          "title": "Flight to Denver",
          "startTime": "2018-01-14T11:45:00-05:00",
          "priority": "HIGH",
          "duration": 257,
          "creationTime": "2017-02-19T15:21:58.547Z"
        },
        {
          "id": "e28a591b-cdc9-4328-9e79-9e4ed60ae7d2",
          "title": "Hotel Von",
          "startTime": "2018-01-14T15:00:00-05:00",
          "priority": "MEDIUM",
          "duration": 4140,
          "creationTime": "2017-02-19T15:21:58.547Z"
        }
      ]
    }
  ]
 }
}
```

Distinct Fields

But, different `Plan` implementations also have distinct fields:

- `Trip` has that `plans` field, among others
- `Lodging` has `address`, for where we are staying
- `Flight` has several additional fields, like `airlineCode` and `flightNumber` to identify the flight

We could just try asking for `airlineCode` all the time:

```
query getAllTrips {
  allTrips {
    ...planFields
    plans {
      ...planFields
      airlineCode
    }
  }
}

fragment planFields on Plan {
  id
  title
  startTime
  priority
  duration
  creationTime
}
```

However, this results in an error, since not every `Plan` has an `airlineCode`:

```
{
  "errors": [
    {
      "message": "Cannot query field \"airlineCode\" on type \"Plan\". Did you mean
to use an inline fragment on \"Flight\"?",
      "locations": [
        {
          "line": 6,
          "column": 7
        }
      ]
    }
  ]
}
```

So, we need some way to tell the GraphQL server to give us `airlineCode`, but only
for `Flight` objects, not for `Trip` and `Lodging` objects. That is where inline fragments
come into play, as is hinted at by the error message.

The "inline" of "inline fragments" refers to having the fragment be defined directly in
the query or mutation, rather than being declared separately the way that
`planFields` is in the GraphQL documents shown earlier in this chapter.

We could rewrite that first GraphQL document to use an inline fragment:

```
query getAllTrips {
  allTrips {
    ... on Plan {
      id
      title
      startTime
      priority
      duration
      creationTime
    }
    plans {
      ... on Plan {
        id
        title
        startTime
        priority
        duration
        creationTime
      }
    }
  }
}
```

Note that inline fragments are not named, akin to how lambdas or anonymous inner classes in Java are not named. The JSON output does not change, because the fragment names never appear in the JSON output anyway. However, this does not really solve our problem, in that we are still limiting ourselves to common fields on `Plan`. Plus, we are duplicating field references now.

The key to inline fragments, for use with interfaces and unions, is that for objects not matching the fragment's type, that fragment is merely skipped, rather than generating an error.

So, for example, we can grab the unique fields from `Flight` using an inline fragment, affecting only those plans that are `Flight` objects:

```
query getAllTrips {
  allTrips {
    ...planFields
    plans {
      ...planFields
      ... on Flight {
        airlineCode
        flightNumber
        departingAirport
        arrivingAirport
```

150

```
      }
    }
  }
}

fragment planFields on Plan {
  id
  title
  startTime
  priority
  duration
  creationTime
}
```

Here, for any object in `plans` that is a `Flight`, in addition to the `planFields`, we *also* retrieve four fields defined on `Flight`. For any object in `plans` that is *not* a `Flight`, the `Flight` inline fragment is ignored. The resulting JSON gives us a mix of objects with full `Flight` details and those with the `planFields` subset:

```
{
  "data": {
    "allTrips": [
      {
        "id": "2c494055-78bc-430c-9ab7-19817f3fc060",
        "title": "Vacation!",
        "startTime": "2017-12-20T13:14:00-05:00",
        "priority": "MEDIUM",
        "duration": 10080,
        "creationTime": "2017-02-19T15:21:58.547Z",
        "plans": [
          {
            "id": "319185bd-fab0-49e3-86ce-251d2aaa5d23",
            "title": "Flight to Chicago",
            "startTime": "2017-12-20T13:14:00-05:00",
            "priority": "HIGH",
            "duration": 150,
            "creationTime": "2017-02-19T15:21:58.547Z",
            "airlineCode": "UAL",
            "flightNumber": "321",
            "departingAirport": "EWR",
            "arrivingAirport": "ORD"
          },
          {
            "id": "319185bd-fab0-49e3-86ce-251d2aaa5d23",
            "title": "House of Munster",
            "startTime": "2017-12-20T15:00:00-05:00",
            "priority": "MEDIUM",
```

```json
          "duration": 9900,
          "creationTime": "2017-02-19T15:21:58.547Z"
        }
      ]
    },
    {
      "id": "e323fed5-6805-4bcf-8cb6-8b7a5014a9d9",
      "title": "Business Trip",
      "startTime": "2018-01-14T11:45:00-05:00",
      "priority": "HIGH",
      "duration": 4320,
      "creationTime": "2017-02-19T15:21:58.547Z",
      "plans": [
        {
          "id": "d40eb2e7-3211-422e-858c-403cbe3fa680",
          "title": "Flight to Denver",
          "startTime": "2018-01-14T11:45:00-05:00",
          "priority": "HIGH",
          "duration": 257,
          "creationTime": "2017-02-19T15:21:58.547Z",
          "airlineCode": "UAL",
          "flightNumber": "456",
          "departingAirport": "EWR",
          "arrivingAirport": "IAD"
        },
        {
          "id": "e28a591b-cdc9-4328-9e79-9e4ed60ae7d2",
          "title": "Hotel Von",
          "startTime": "2018-01-14T15:00:00-05:00",
          "priority": "MEDIUM",
          "duration": 4140,
          "creationTime": "2017-02-19T15:21:58.547Z"
        }
      ]
    }
  ]
 }
}
```

There is no limit to the number of inline fragments that you use, though the type of the inline fragment has to be a candidate for the field in question. In the case of `plans`, we can have inline fragments based on `Plan` or any type that implements the `Plan` interface, but we cannot have inline fragments for something that does not implement `Plan`, such as `Comment`:

```
query getAllTrips {
  allTrips {
```

```
      ...planFields
    plans {
      ...planFields
      ... on Flight {
        airlineCode
        flightNumber
        departingAirport
        arrivingAirport
      }
      ... on Comment {
        text
      }
    }
  }
}

fragment planFields on Plan {
  id
  title
  startTime
  priority
  duration
  creationTime
}
```

This results in an error:

```
{
  "errors": [
    {
      "message": "Fragment cannot be spread here as objects of type \"Plan\" can
never be of type \"Comment\".",
      "locations": [
        {
          "line": 12,
          "column": 7
        }
      ]
    }
  ]
}
```

But we are welcome to get distinct fields both for `Flight` objects and for `Lodging` objects:

```
query getAllTrips {
  allTrips {
```

```
      ...planFields
    plans {
      ...planFields
      ... on Flight {
        airlineCode
        flightNumber
        departingAirport
        arrivingAirport
      }
      ... on Lodging {
        address
      }
    }
  }
}

fragment planFields on Plan {
  id
  title
  startTime
  priority
  duration
  creationTime
}
```

Now, Lodging objects give us address fields:

```
{
  "data": {
    "allTrips": [
      {
        "id": "2c494055-78bc-430c-9ab7-19817f3fc060",
        "title": "Vacation!",
        "startTime": "2017-12-20T13:14:00-05:00",
        "priority": "MEDIUM",
        "duration": 10080,
        "creationTime": "2017-02-19T15:21:58.547Z",
        "plans": [
          {
            "id": "319185bd-fab0-49e3-86ce-251d2aaa5d23",
            "title": "Flight to Chicago",
            "startTime": "2017-12-20T13:14:00-05:00",
            "priority": "HIGH",
            "duration": 150,
            "creationTime": "2017-02-19T15:21:58.547Z",
            "airlineCode": "UAL",
            "flightNumber": "321",
```

```
            "departingAirport": "EWR",
            "arrivingAirport": "ORD"
          },
          {
            "id": "319185bd-fab0-49e3-86ce-251d2aaa5d23",
            "title": "House of Munster",
            "startTime": "2017-12-20T15:00:00-05:00",
            "priority": "MEDIUM",
            "duration": 9900,
            "creationTime": "2017-02-19T15:21:58.547Z",
            "address": "1313 Mockingbird Lane, Springfield, IL, USA 62701"
          }
        ]
      },
      {
        "id": "e323fed5-6805-4bcf-8cb6-8b7a5014a9d9",
        "title": "Business Trip",
        "startTime": "2018-01-14T11:45:00-05:00",
        "priority": "HIGH",
        "duration": 4320,
        "creationTime": "2017-02-19T15:21:58.547Z",
        "plans": [
          {
            "id": "d40eb2e7-3211-422e-858c-403cbe3fa680",
            "title": "Flight to Denver",
            "startTime": "2018-01-14T11:45:00-05:00",
            "priority": "HIGH",
            "duration": 257,
            "creationTime": "2017-02-19T15:21:58.547Z",
            "airlineCode": "UAL",
            "flightNumber": "456",
            "departingAirport": "EWR",
            "arrivingAirport": "IAD"
          },
          {
            "id": "e28a591b-cdc9-4328-9e79-9e4ed60ae7d2",
            "title": "Hotel Von",
            "startTime": "2018-01-14T15:00:00-05:00",
            "priority": "MEDIUM",
            "duration": 4140,
            "creationTime": "2017-02-19T15:21:58.547Z",
            "address": "10 Backfield Place, Denver, CO 81023"
          }
        ]
      }
    ]
  }
}
```

In this fashion, we can get whatever fields we need, from whatever types might be in `plans`. In some cases, we may not need those distinct fields right away (e.g., building a list of the plans), so we grab the common fields on `Plan` and that's it.

Unions

Working with unions is much the same as working with interfaces, except that there are no common fields. You have to use inline fragments for everything that you want to retrieve.

As was noted in a previous chapter, a `Plan` has a `notes` field, of type `Note`. A `Note` is a `Union`, consisting of two possible types: `Comment` and `Link`:
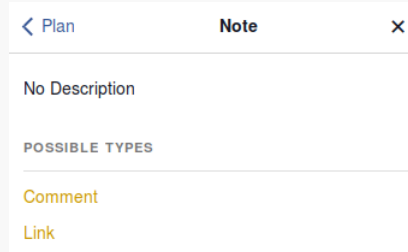
Figure 35: GraphiQL Documentation, Showing Note
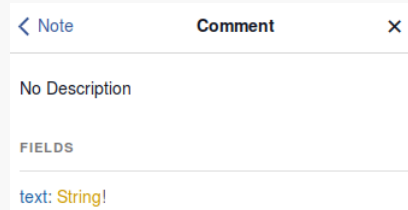
Figure 36: GraphiQL Documentation, Showing Comment
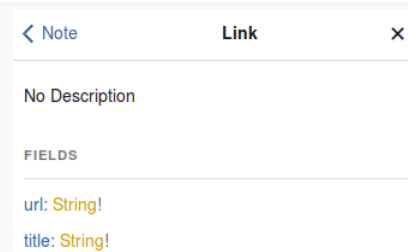
Figure 37: GraphiQL Documentation, Showing Link

To be able to retrieve anything from `notes`, we need to use inline fragments, as there

are no fields in common between `Comment` and `Link`:

```
query getAllTrips {
  allTrips {
    ...planFields
    notes {
      ... on Comment {
        text
      }
      ... on Link {
        url
        title
      }
    }
  }
}

fragment planFields on Plan {
  id
  title
  startTime
  priority
  duration
  creationTime
}
```

Here, we are retrieving the `notes`, specifically the `text` field from any `Comment` and the `url` and `title` fields from any `Link`. For trips with notes, we get those results:

```
{
  "data": {
    "allTrips": [
      {
        "id": "2c494055-78bc-430c-9ab7-19817f3fc060",
        "title": "Vacation!",
        "startTime": "2017-12-20T13:14:00-05:00",
        "priority": "MEDIUM",
        "duration": 10080,
        "creationTime": "2017-02-19T15:21:58.547Z",
        "notes": [
          {
            "text": "It's gonna be great!"
          },
          {
            "url": "http://www.miragrant.com/",
            "title": "Source of some reading material for the trip"
          }
```

```
        ]
      },
      {
        "id": "e323fed5-6805-4bcf-8cb6-8b7a5014a9d9",
        "title": "Business Trip",
        "startTime": "2018-01-14T11:45:00-05:00",
        "priority": "HIGH",
        "duration": 4320,
        "creationTime": "2017-02-19T15:21:58.547Z",
        "notes": null
      }
    ]
  }
}
```

Interfaces, Unions, and Apollo-Android

Apollo-Android handles interfaces and unions, but it suffers from some of the same design decisions that drive you towards using fragments more than you might have planned.

For example, suppose you try using this GraphQL document with Apollo-Android, such as in one of the book's `Trips/CW/` sample apps:

```
query getAllTrips {
  allTrips {
    ...planFields
    plans {
      ...planFields
      ... on Flight {
        airlineCode
        flightNumber
        departingAirport
        arrivingAirport
      }
      ... on Lodging {
        address
      }
    }
    notes {
      ... on Comment {
        text
      }
      ... on Link {
        url
```

```
        title
      }
    }
  }
}

query findTrips($search: String!) {
  findTrips(searchFor: $search) {
    ...planFields
    plans {
      ...planFields
      ... on Flight {
        airlineCode
        flightNumber
        departingAirport
        arrivingAirport
      }
      ... on Lodging {
        address
      }
    }
    notes {
      ... on Comment {
        text
      }
      ... on Link {
        url
        title
      }
    }
  }
}

fragment planFields on Plan {
  id
  title
  startTime
  priority
  duration
  creationTime
}
```

The resulting `GetAllTrips.Data.AllTrip` and `FindTrips.Data.FindTrip` classes have `plans()` and `notes()` fields. These are a `List` on `Plan` and `Note` classes, respectively. However, each operation gets its own definition of `Plan` and `Note`:

Operation Class	Plan Class	Note Class
GetAllTrips	GetAllTrips.Data.AllTrips.Plan	GetAllTrips.Data.AllTrips.Note
FindTrips	FindTrips.Data.FindTrip.Plan	FindTrips.Data.FindTrip.Note

Those classes are unrelated, requiring you to have unique logic for each.

To access the `planFields`, you then have to call `fragments().planFields()` on a member of the `plans()` list. This follows the `fragments()` pattern from before and is not terribly surprising.

The inline fragment fields, though, get put into yet another code-generated class: `As...`, where the `...` is replaced by the type referenced by the inline fragment. So, for example, for the `plans` field, where we used inline fragments for `Flight` and `Lodging`, we get `AsFlight` and `AsLodging` classes (and, again, one per operation, such as `FindTrips.Data.FindTrip.Plan.AsFlight`). You get an instance of those via `asFlight()` and `asLodging()` methods on the associated `Plan` object. On the `AsFlight` and `AsLodging` objects, you have getter methods for their specific fields (e.g., `airlineCode()`), plus their own `fragments().planFields()` for accessing the common fields.

Alternatively, you can move all of that stuff into its own common fragment:

```
query getAllTrips {
  allTrips {
    ...tripFields
  }
}

query findTrips($search: String!) {
  findTrips(searchFor: $search) {
    ...tripFields
  }
}

fragment planFields on Plan {
  id
  title
  startTime
  priority
  duration
```

```
  creationTime
}

fragment tripFields on Trip {
  ...planFields
  plans {
    ...planFields
    ... on Flight {
      airlineCode
      flightNumber
      departingAirport
      arrivingAirport
    }
    ... on Lodging {
      address
    }
  }
  notes {
    ... on Comment {
      text
    }
    ... on Link {
      url
      title
    }
  }
}
```

(from Trips/CW/StaticInlineFrag/app/src/main/graphql/com/commonsware/graphql/trips/api/TripServer.graphql)

Now, there is a single definition of `TripFields`, with its own `TripFields.Note` and `TripFields.Plan` classes. You still wind up with the `As...` classes and `as...()` methods (e.g., `AsFlight` and `asFlight()`) to access the fields from the inline fragments. However, at least the types are common between the two operations, allowing you to handle both GraphQL responses with (mostly) the same Java code.

Miscellaneous GraphQL Syntax

Many programming and data-access languages have bits and pieces, or odds and sods, of miscellaneous syntax that are important, yet not important enough to warrant their own chapter in a book.

This chapter covers some of that syntax, with respect to GraphQL.

Arguments on Nested Fields

Back in the chapter on variables and arguments, we saw how root fields can take arguments, to make it easier to pass in details that may vary by invocation:

```
query find($search: String!) {
  findTrip(searchFor: $search) {
    id
    title
    startTime
    priority
    duration
  }
}
```

Here, the argument is applied to one of the root query fields, `findTrip`. This feels fairly typical and a bit reminiscent of passing parameter values to Java methods.

However, fields other than root fields might also accept arguments. These can have the same sorts of roles as do arguments on root fields: sort order, pagination, etc. Here are some other scenarios where arguments on nested fields may prove useful.

163

Scenario: Conversion

Sometimes, the argument might be for simple things like format or unit conversion:

```
query find($search: String!) {
  findTrip(searchFor: $search) {
    id
    title
    startTime
    priority
    duration(timeUnit: DAYS)
  }
}
```

Here, a (theoretical) revised version of our server takes a `timeUnit` argument on `duration`, from some (theoretical) enum where one of the values is `DAYS`. This would be akin to using `TimeUnit.DAYS` in Java date/time conversions.

Typically — though not universally — fields in this scenario will have a default value for the argument. That way, you can either include the argument or skip it, as you see fit. If you skip the argument, some default format or unit will be used (e.g., minutes in the case of `duration`).

Scenario: Filtering

Sometimes, the argument might be for restricting the data to some subset:

```
query find($search: String!) {
  findTrip(searchFor: $search) {
    id
    title
    startTime
    priority
    duration(timeUnit: DAYS)
    plans(type: FLIGHT) {
      id
      title
    }
  }
}
```

Here, another (theoretical) revised version of the server takes a `type` argument on `plans`, for some (theoretical) enum where one of the values is `FLIGHT`. This might be used by the server to only return those plans that are flights, not lodging stays, for

clients that only care about the flights.

Once again, often times the argument will have a default value; in the case of filtering, the typical default meaning is "give me everything".

Developing Clients Using Arguments on Nested Fields

In general, though, the fact that these arguments are nested does not really impact your development of a GraphQL client. Either the arguments are being set from variables, or they are not. Variables are still all declared on the operation itself. So, you might have more variables than before, but the fact that one or more of those variables happen to be used in nested fields' arguments does not impact how you provide the variables' values:

- in dynamic GraphQL, via the `variables` JSON object in the request
- in Apollo-Android, via the `Variables` class code-generated from your GraphQL document and the associated schema

Directives

In Java, we have annotations. These are denoted by an `@` prefix in front of some symbol, such as `@Override` or `@TargetApi` or `@Subscribe`. This is an extension point for the Java language itself, in effect. Not only can Java provide annotations (e.g., `@Override`), and not only can Android add more annotations (e.g., `@TargetApi`), but third-party libraries can define annotations (e.g., `@Subscribe` from [greenrobot's EventBus](#)). The Java programming language itself provides hooks for libraries to declare available annotations and find out what annotations a developer used (via an annotation processor). However, Java itself does not define what most of the annotations actually *are*.

In GraphQL, directives fill a similar role. As it turns out, the syntax is also a bit reminiscent of Java annotations, with `@` as a prefix before some symbol.

@include and @skip

Two directives are defined at present in the GraphQL specification: `@include` and `@skip`. These are opposites:

- A field denoted with `@include` means "include this field if and only if a certain variable is `true`"

165

- A field denoted with @skip means "include this field if and only if a certain variable is false"

For example, you might want to use the same GraphQL document for requesting either just the basic trip data or all of its nested plans as well. The server could offer that via dedicated server-side logic:

```
query all($getPlans: Boolean!) {
  allTrips {
    id
    title
    startTime
    priority
    duration
    plans(sockItToMe: $getPlans) {
      id
      title
    }
  }
}
```

Here, the (theoretical) sockItToMe argument would be used by the server to determine whether to return the plans or return something else (null or an empty list).

However, GraphQL supports this without custom server logic:

```
query all($getPlans: Boolean!) {
  allTrips {
    id
    title
    startTime
    priority
    duration
    plans @include(if: $getPlans) {
      id
      title
    }
  }
}
```

If you execute this in GraphiQL against the public demo GraphQL server, and you set the getPlans variable to true, you get the nested plans. However, if you set the getPlans variable to false, the plans field is ignored and is not returned.

`@skip` just inverts the role of the boolean variable:

```
query all($noPlans: Boolean!) {
  allTrips {
    id
    title
    startTime
    priority
    duration
    plans @skip(if: $noPlans) {
      id
      title
    }
  }
}
```

Here, if `noPlans` is `true`, we skip `plans`; if `noPlans` is `false`, we include the `plans`.

From the standpoint of your GraphQL client, other than having additional variables for the `if` argument on the `@include` or `@skip` directives, there is no effect on your code. These variables are indistinguishable from any other variables that you might be using.

Custom Directives

In theory, you may have access to more directives than this:

> Server implementations may also add experimental features by defining completely new directives.

(from [the GraphQL documentation](#)).

GraphQL itself may gain more directives in the future as well:

> As future versions of GraphQL adopt new configurable execution capabilities, they may be exposed via directives.

(from [the GraphQL specification](#))

The [introspection API](#) will supply details of the available directives, and so they should be available in tools like GraphiQL.

Deprecations

On the whole, GraphQL is relatively resilient to change. In particular, a GraphQL server can add new fields — both top-level and fields on specific types — without impacting any existing client of that server. Since clients ask for the specific fields that they want, they will never be given the new fields, until such time as they start asking for them.

However, the reverse is not as smooth. If a GraphQL server removes fields, it immediately breaks any clients that continue requesting them.

As a result, a server can mark a field, or the value of an enum, as deprecated. This is akin to having a Java class or method be marked as deprecated: you can still use it, but the tools will warn you about the deprecation. So, for example, GraphiQL can hide deprecated elements, or render them in a different style (e.g., strikethrough notation, akin to how Android Studio denotes deprecated Java elements).

Introspection

When we query a GraphQL server, we request root fields and sub-fields of those roots, sub-sub-fields of those sub-fields, and so forth, creating a tree of results.

The fields that we have focused on to date have been data: trips, plans, and things like that. However, GraphQL also has another set of fields representing metadata about what we are querying. This set of fields allows tools like GraphiQL to know the nature of what the server understands, so the tools can guide you, generate code for you (as with Apollo-Android), and so forth. Those fields can also be useful to you as part of your normal GraphQL requests, to give you some context around the response that you get.

These metadata fields form GraphQL's introspection system, which we will focus on in this chapter.

Adding a Type To Your Response

In a previous chapter, we examined how to query on [interfaces and unions](). However, savvy developers will have noticed a bit of a hole in our query and its response: we do not get any information about what we are getting back.

For example, we had this GraphQL document:

```
query getAllTrips {
  allTrips {
    ...planFields
    plans {
      ...planFields
    }
  }
```

169

```
}

fragment planFields on Plan {
  id
  title
  startTime
  priority
  duration
  creationTime
}
```

That, in turn, generates this JSON response:

```
{
  "data": {
    "allTrips": [
      {
        "id": "2c494055-78bc-430c-9ab7-19817f3fc060",
        "title": "Vacation!",
        "startTime": "2017-12-20T13:14:00-05:00",
        "priority": "MEDIUM",
        "duration": 10080,
        "creationTime": "2017-02-19T15:21:58.547Z",
        "plans": [
          {
            "id": "319185bd-fab0-49e3-86ce-251d2aaa5d23",
            "title": "Flight to Chicago",
            "startTime": "2017-12-20T13:14:00-05:00",
            "priority": "HIGH",
            "duration": 150,
            "creationTime": "2017-02-19T15:21:58.547Z"
          },
          {
            "id": "319185bd-fab0-49e3-86ce-251d2aaa5d23",
            "title": "House of Munster",
            "startTime": "2017-12-20T15:00:00-05:00",
            "priority": "MEDIUM",
            "duration": 9900,
            "creationTime": "2017-02-19T15:21:58.547Z"
          }
        ]
      },
      {
        "id": "e323fed5-6805-4bcf-8cb6-8b7a5014a9d9",
        "title": "Business Trip",
        "startTime": "2018-01-14T11:45:00-05:00",
        "priority": "HIGH",
```

```
        "duration": 4320,
        "creationTime": "2017-02-19T15:21:58.547Z",
        "plans": [
          {
            "id": "d40eb2e7-3211-422e-858c-403cbe3fa680",
            "title": "Flight to Denver",
            "startTime": "2018-01-14T11:45:00-05:00",
            "priority": "HIGH",
            "duration": 257,
            "creationTime": "2017-02-19T15:21:58.547Z"
          },
          {
            "id": "e28a591b-cdc9-4328-9e79-9e4ed60ae7d2",
            "title": "Hotel Von",
            "startTime": "2018-01-14T15:00:00-05:00",
            "priority": "MEDIUM",
            "duration": 4140,
            "creationTime": "2017-02-19T15:21:58.547Z"
          }
        ]
      }
    ]
  }
}
```

We get back the fields that we ask for. However, there is nothing to tell us what the underlying type is behind the objects in the `plans` array. As humans, we can tell that the "Flight to Denver" `Plan` probably is a `Flight`, as that would be a rather strange name for some `Lodging`. However, we cannot readily teach software to make that distinction, nor should we need to.

Using inline fragments can help a little:

```
query getAllTrips {
  allTrips {
    ...planFields
    plans {
      ...planFields
      ... on Flight {
        airlineCode
        flightNumber
        departingAirport
        arrivingAirport
      }
      ... on Lodging {
        address
```

```
        }
      }
    }
  }
}

fragment planFields on Plan {
  id
  title
  startTime
  priority
  duration
  creationTime
}
```

Now, we could say that if a `Plan` has an `airlineCode`, then it is a `Flight`, whereas if it has an `address`, then it is a `Lodging`. However, we are making an inference of the underlying types based on the fields that we get back. Wouldn't it be nice if the JSON just said, plainly, what each of those objects are?

The good news is that we can get that, by adding a single additional field to `planFields: __typename`. This provides to us the type name of whatever object it is that we are obtaining fields from. So, by changing `planFields` to:

```
fragment planFields on Plan {
  __typename
  id
  title
  startTime
  priority
  duration
  creationTime
}
```

...we now get output like this:

```
{
  "data": {
    "allTrips": [
      {
        "__typename": "Trip",
        "id": "2c494055-78bc-430c-9ab7-19817f3fc060",
        "title": "Vacation!",
        "startTime": "2017-12-20T13:14:00-05:00",
        "priority": "MEDIUM",
        "duration": 10080,
        "creationTime": "2017-02-19T15:21:58.547Z",
```

```
      "plans": [
       {
         "__typename": "Flight",
         "id": "319185bd-fab0-49e3-86ce-251d2aaa5d23",
         "title": "Flight to Chicago",
         "startTime": "2017-12-20T13:14:00-05:00",
         "priority": "HIGH",
         "duration": 150,
         "creationTime": "2017-02-19T15:21:58.547Z"
       },
       {
         "__typename": "Lodging",
         "id": "319185bd-fab0-49e3-86ce-251d2aaa5d23",
         "title": "House of Munster",
         "startTime": "2017-12-20T15:00:00-05:00",
         "priority": "MEDIUM",
         "duration": 9900,
         "creationTime": "2017-02-19T15:21:58.547Z"
       }
      ]
     },
     {
       "__typename": "Trip",
       "id": "e323fed5-6805-4bcf-8cb6-8b7a5014a9d9",
       "title": "Business Trip",
       "startTime": "2018-01-14T11:45:00-05:00",
       "priority": "HIGH",
       "duration": 4320,
       "creationTime": "2017-02-19T15:21:58.547Z",
       "plans": [
        {
          "__typename": "Flight",
          "id": "d40eb2e7-3211-422e-858c-403cbe3fa680",
          "title": "Flight to Denver",
          "startTime": "2018-01-14T11:45:00-05:00",
          "priority": "HIGH",
          "duration": 257,
          "creationTime": "2017-02-19T15:21:58.547Z"
        },
        {
          "__typename": "Lodging",
          "id": "e28a591b-cdc9-4328-9e79-9e4ed60ae7d2",
          "title": "Hotel Von",
          "startTime": "2018-01-14T15:00:00-05:00",
          "priority": "MEDIUM",
          "duration": 4140,
          "creationTime": "2017-02-19T15:21:58.547Z"
        }
```

173

```
      ]
    }
  ]
  }
}
```

Because we added __typename to planFields, everywhere we are getting fields from a Plan — both the Trip and its plans — we get a __typename telling us how to interpret the data. So, now we have a positive indicator of what is a Flight and what is Lodging, instead of having to infer the type from the available fields.

Note that you can alias __typename, just as you can alias any other sort of field:

```
fragment planFields on Plan {
  type: __typename
  id
  title
  startTime
  priority
  duration
  creationTime
}
```

Now, we would get the type as a type field in the JSON, rather than as __typename.

Note that Apollo-Android does not allow you to retrieve the __typename field. Apollo-Android uses that internally for type resolution — this is how Apollo-Android can create Flight and Lodging objects correctly. However, as a result, if you try to add __typename yourself to a query, your build will fail. However, your result objects (e.g., AllTrip) have a __typename() method that will return the __typename value as retrieved by Apollo-Android.

Introspection Beyond the Type Name

Getting the type name may be something of use to many apps (or, at least those not using Apollo-Android).

Introspection of a GraphQL server beyond the type name is something that a few apps will use, mostly in the area of development tools:

- Code generators, like Apollo-Android
- GraphQL interactive clients, like GraphiQL

- Documentation tools (e.g., schema graphing tools)
- Middleware layers (e.g., REST-to-GraphQL converter)
- And so on

Let's see how those sorts of tools can examine the server's GraphQL schema via introspection queries.

Requesting the Roots

In the GrapiQL for the public demo GraphQL server (https://graphql-demo.commonsware.com/0.3/graphql), try entering the following query:

```
{
  __schema {
    queryType {
      name
    }
    mutationType {
      name
    }
  }
}
```

Anything beginning with __ is part of the introspection system. In particular, querying the __schema does what it says: it queries the schema to find out what is inside of it.

Here, we are requesting two fields: queryType and mutationType. Those are the GraphQL types that have been designated by the server as defining the root fields available for queries and mutations, respectively.

A typical implementation will have those types named Query and Mutation, respectively, which we get back by asking for the name fields of those types:

```
{
  "data": {
    "__schema": {
      "queryType": {
        "name": "Query"
      },
      "mutationType": {
        "name": "Mutation"
      }
    }
```

175

```
  }
}
```

Listing the Types

You can get a roster of all the registered types in the schema by querying on the types field:

```
{
  __schema {
    types {
      name
    }
  }
}
```

This will return more than you might expect, though:

```
{
  "data": {
    "__schema": {
      "types": [
        {
          "name": "Query"
        },
        {
          "name": "Trip"
        },
        {
          "name": "Plan"
        },
        {
          "name": "ID"
        },
        {
          "name": "String"
        },
        {
          "name": "Note"
        },
        {
          "name": "Comment"
        },
        {
          "name": "Link"
        },
```

```
        {
          "name": "Priority"
        },
        {
          "name": "Int"
        },
        {
          "name": "Mutation"
        },
        {
          "name": "TripInput"
        },
        {
          "name": "__Schema"
        },
        {
          "name": "__Type"
        },
        {
          "name": "__TypeKind"
        },
        {
          "name": "Boolean"
        },
        {
          "name": "__Field"
        },
        {
          "name": "__InputValue"
        },
        {
          "name": "__EnumValue"
        },
        {
          "name": "__Directive"
        },
        {
          "name": "__DirectiveLocation"
        },
        {
          "name": "Lodging"
        },
        {
          "name": "Flight"
        }
      ]
    }
}
```

177

```
}
```

The registered types can be divided into three groups:

- Those that are part of the introspection system, identifiable by their leading double-underscores (e.g., `__Type`)
- Those that are part of the actual GraphQL that the server-side developer wrote in support of this Web service
- Those that are part of standard GraphQL, or other types exposed by the GraphQL framework that the server uses, that are referenced by the specific schema used by the Web service

In the JSON above — pulled from the GraphQL demo server — the following types are standard GraphQL ones:

- `Boolean`
- `ID`
- `Int`
- `Float`

Those appear in the output because they are directly referenced by the GraphQL schema used by the server. Other stock types, such as `Float`, do not appear. And, given that GraphQL is type-safe, any unused type is unusable — there is nothing in the server's schema that can accept a `Float`.

We can get more information about these types by adding in two more fields to our query, defined on `__Type`: `kind` and `description`:

```
{
  __schema {
    types {
      name
      kind
      description
    }
  }
}
```

The `description` is an optional comment about what the type is for. The `kind` indicates the general type of GraphQL type that this is: `SCALAR`, `OBJECT`, `ENUM`, etc.

The GraphQL demo server will return something akin to:

```
{
  "data": {
    "__schema": {
      "types": [
        {
          "name": "Query",
          "kind": "OBJECT",
          "description": "These are the available queries, representing data that  we
can retrieve from this server"
        },
        {
          "name": "Trip",
          "kind": "OBJECT",
          "description": "Represents a collection of plans encompassing some trip to
somewhere for something"
        },
        {
          "name": "Plan",
          "kind": "INTERFACE",
          "description": ""
        },
        {
          "name": "ID",
          "kind": "SCALAR",
          "description": "The `ID` scalar type represents a unique identifier, often
used to refetch an object or as key for a cache. The ID type appears in a JSON
response as a String; however, it is not intended to be human-readable. When expected
as an input type, any string (such as `\"4\"`) or integer (such as `4`) input value
will be accepted as an ID."
        },
        {
          "name": "String",
          "kind": "SCALAR",
          "description": "The `String` scalar type represents textual data,
represented as UTF-8 character sequences. The String type is most often used by
GraphQL to represent free-form human-readable text."
        },
        {
          "name": "Note",
          "kind": "UNION",
          "description": ""
        },
        {
          "name": "Comment",
          "kind": "OBJECT",
          "description": ""
        },
        {
```

179

```
          "name": "Link",
          "kind": "OBJECT",
          "description": ""
        },
        {
          "name": "Priority",
          "kind": "ENUM",
          "description": ""
        },
        {
          "name": "Int",
          "kind": "SCALAR",
          "description": "The `Int` scalar type represents non-fractional signed
whole numeric values. Int can represent values between -(2^31) and 2^31 - 1. "
        },
        {
          "name": "Mutation",
          "kind": "OBJECT",
          "description": ""
        },
        {
          "name": "TripInput",
          "kind": "INPUT_OBJECT",
          "description": ""
        },
        {
          "name": "__Schema",
          "kind": "OBJECT",
          "description": "A GraphQL Schema defines the capabilities of a GraphQL
server. It exposes all available types and directives on the server, as well as the
entry points for query, mutation, and subscription operations."
        },
        {
          "name": "__Type",
          "kind": "OBJECT",
          "description": "The fundamental unit of any GraphQL Schema is the type.
There are many kinds of types in GraphQL as represented by the `__TypeKind` enum.\n\
nDepending on the kind of a type, certain fields describe information about that
type. Scalar types provide no information beyond a name and description, while Enum
types provide their values. Object and Interface types provide the fields they
describe. Abstract types, Union and Interface, provide the Object types possible at
runtime. List and NonNull types compose other types."
        },
        {
          "name": "__TypeKind",
          "kind": "ENUM",
          "description": "An enum describing what kind of type a given `__Type` is."
        },
```

```
      {
        "name": "Boolean",
        "kind": "SCALAR",
        "description": "The `Boolean` scalar type represents `true` or `false`."
      },
      {
        "name": "__Field",
        "kind": "OBJECT",
        "description": "Object and Interface types are described by a list of
Fields, each of which has a name, potentially a list of arguments, and a return type."
      },
      {
        "name": "__InputValue",
        "kind": "OBJECT",
        "description": "Arguments provided to Fields or Directives and the input
fields of an InputObject are represented as Input Values which describe their type
and optionally a default value."
      },
      {
        "name": "__EnumValue",
        "kind": "OBJECT",
        "description": "One possible value for a given Enum. Enum values are unique
values, not a placeholder for a string or numeric value. However an Enum value is
returned in a JSON response as a string."
      },
      {
        "name": "__Directive",
        "kind": "OBJECT",
        "description": "A Directive provides a way to describe alternate runtime
execution and type validation behavior in a GraphQL document.\n\nIn some cases, you
need to provide options to alter GraphQL's execution behavior in ways field arguments
will not suffice, such as conditionally including or skipping a field. Directives
provide this by describing additional information to the executor."
      },
      {
        "name": "__DirectiveLocation",
        "kind": "ENUM",
        "description": "A Directive can be adjacent to many parts of the GraphQL
language, a __DirectiveLocation describes one such possible adjacencies."
      },
      {
        "name": "Lodging",
        "kind": "OBJECT",
        "description": ""
      },
      {
        "name": "Flight",
        "kind": "OBJECT",
```

```
        "description": ""
      }
    ]
  }
 }
}
```

Collecting the Fields

The fields for a type can be obtained by querying the `fields` field on the `__Type`:

```
{
  __schema {
    types {
      name
      fields {
        name
      }
    }
  }
}
```

This just gives us each field's name (here, just showing `Trip` for brevity):

```
{
  "name": "Trip",
  "fields": [
    {
      "name": "id"
    },
    {
      "name": "startTime"
    },
    {
      "name": "title"
    },
    {
      "name": "notes"
    },
    {
      "name": "creationTime"
    },
    {
      "name": "updateTime"
    },
    {
```

```
      "name": "priority"
    },
    {
      "name": "duration"
    },
    {
      "name": "plans"
    }
  ]
}
```

This, of course, just gives us the field name. We can also ask for its `description`, but we really need more to go on than those.

Field Types

A field has a `type` field, that we can use to find out the type of that field... at least partially:

```
{
  __schema {
    types {
      name
      fields {
        name
        type {
          name
          kind
        }
      }
    }
  }
}
```

That gives us the following `Trip` result:

```
{
  "name": "Trip",
  "fields": [
    {
      "name": "id",
      "type": {
        "name": null,
        "kind": "NON_NULL"
      }
    },
```

```
{
  "name": "startTime",
  "type": {
    "name": null,
    "kind": "NON_NULL"
  }
},
{
  "name": "title",
  "type": {
    "name": null,
    "kind": "NON_NULL"
  }
},
{
  "name": "notes",
  "type": {
    "name": null,
    "kind": "LIST"
  }
},
{
  "name": "creationTime",
  "type": {
    "name": null,
    "kind": "NON_NULL"
  }
},
{
  "name": "updateTime",
  "type": {
    "name": null,
    "kind": "NON_NULL"
  }
},
{
  "name": "priority",
  "type": {
    "name": null,
    "kind": "NON_NULL"
  }
},
{
  "name": "duration",
  "type": {
    "name": null,
    "kind": "NON_NULL"
  }
```

```
    },
    {
      "name": "plans",
      "type": {
        "name": null,
        "kind": "NON_NULL"
      }
    }
  ]
}
```

The catch is that all of these types are not-null or list types. GraphQL handles this via type decoration, wrapping a type in a NOT_NULL or LIST type. For those, ofType unwraps the next layer down:

```
{
  __schema {
    types {
      name
      fields {
        name
        type {
          name
          kind
          ofType {
            name
            kind
          }
        }
      }
    }
  }
}
```

However, that will be insufficient. A non-null list of non-null types requires *four* levels of nesting (the original type and three ofType fields within it):

```
{
  __schema {
    types {
      name
      fields {
        name
        type {
          name
          kind
```

```
        ofType {
          name
          kind
          ofType {
            name
            kind
            ofType {
              name
              kind
            }
          }
        }
      }
    }
  }
}
```

This gives us the following details for `Trip`:

```json
{
  "name": "Trip",
  "fields": [
    {
      "name": "id",
      "type": {
        "name": null,
        "kind": "NON_NULL",
        "ofType": {
          "name": "ID",
          "kind": "SCALAR",
          "ofType": null
        }
      }
    },
    {
      "name": "startTime",
      "type": {
        "name": null,
        "kind": "NON_NULL",
        "ofType": {
          "name": "String",
          "kind": "SCALAR",
          "ofType": null
        }
      }
    },
```

```
    {
      "name": "title",
      "type": {
        "name": null,
        "kind": "NON_NULL",
        "ofType": {
          "name": "String",
          "kind": "SCALAR",
          "ofType": null
        }
      }
    },
    {
      "name": "notes",
      "type": {
        "name": null,
        "kind": "LIST",
        "ofType": {
          "name": null,
          "kind": "NON_NULL",
          "ofType": {
            "name": "Note",
            "kind": "UNION",
            "ofType": null
          }
        }
      }
    },
    {
      "name": "creationTime",
      "type": {
        "name": null,
        "kind": "NON_NULL",
        "ofType": {
          "name": "String",
          "kind": "SCALAR",
          "ofType": null
        }
      }
    },
    {
      "name": "updateTime",
      "type": {
        "name": null,
        "kind": "NON_NULL",
        "ofType": {
          "name": "String",
          "kind": "SCALAR",
```

```
          "ofType": null
        }
      }
    },
    {
      "name": "priority",
      "type": {
        "name": null,
        "kind": "NON_NULL",
        "ofType": {
          "name": "Priority",
          "kind": "ENUM",
          "ofType": null
        }
      }
    },
    {
      "name": "duration",
      "type": {
        "name": null,
        "kind": "NON_NULL",
        "ofType": {
          "name": "Int",
          "kind": "SCALAR",
          "ofType": null
        }
      }
    },
    {
      "name": "plans",
      "type": {
        "name": null,
        "kind": "NON_NULL",
        "ofType": {
          "name": null,
          "kind": "LIST",
          "ofType": {
            "name": null,
            "kind": "NON_NULL",
            "ofType": {
              "name": "Plan",
              "kind": "INTERFACE"
            }
          }
        }
      }
    }
  }
]
```

188

```
},
{
  "name": "Plan",
  "fields": [
    {
      "name": "id",
      "type": {
        "name": null,
        "kind": "NON_NULL",
        "ofType": {
          "name": "ID",
          "kind": "SCALAR",
          "ofType": null
        }
      }
    },
    {
      "name": "startTime",
      "type": {
        "name": null,
        "kind": "NON_NULL",
        "ofType": {
          "name": "String",
          "kind": "SCALAR",
          "ofType": null
        }
      }
    },
    {
      "name": "title",
      "type": {
        "name": null,
        "kind": "NON_NULL",
        "ofType": {
          "name": "String",
          "kind": "SCALAR",
          "ofType": null
        }
      }
    },
    {
      "name": "notes",
      "type": {
        "name": null,
        "kind": "LIST",
        "ofType": {
          "name": null,
          "kind": "NON_NULL",
```

189

```
          "ofType": {
            "name": "Note",
            "kind": "UNION",
            "ofType": null
          }
        }
      }
    },
    {
      "name": "creationTime",
      "type": {
        "name": null,
        "kind": "NON_NULL",
        "ofType": {
          "name": "String",
          "kind": "SCALAR",
          "ofType": null
        }
      }
    },
    {
      "name": "updateTime",
      "type": {
        "name": null,
        "kind": "NON_NULL",
        "ofType": {
          "name": "String",
          "kind": "SCALAR",
          "ofType": null
        }
      }
    },
    {
      "name": "priority",
      "type": {
        "name": null,
        "kind": "NON_NULL",
        "ofType": {
          "name": "Priority",
          "kind": "ENUM",
          "ofType": null
        }
      }
    },
    {
      "name": "duration",
      "type": {
        "name": null,
```

190

```
      "kind": "NON_NULL",
      "ofType": {
        "name": "Int",
        "kind": "SCALAR",
        "ofType": null
      }
    }
  }
  ]
}
```

(in case you had not already determined this... introspection responses can be rather verbose)

That type-unwrapping GraphQL snippet will be needed elsewhere, so you can always define it as a fragment:

```
{
  __schema {
    types {
      name
      fields {
        name
        type {
          ...TypeDesc
        }
      }
    }
  }
}

fragment TypeDesc on __Type {
  name
  kind
  ofType {
    name
    kind
    ofType {
      name
      kind
      ofType {
        name
        kind
      }
    }
  }
}
```

Field Arguments

You can find out about arguments of fields via the `args` field on `__Type`:

```
{
  __schema {
    types {
      name
      fields {
        name
        args {
          name
          description
          type {
            ...TypeDesc
          }
          defaultValue
        }
      }
    }
  }
}

fragment TypeDesc on __Type {
  name
  kind
  ofType {
    name
    kind
    ofType {
      name
      kind
      ofType {
        name
        kind
      }
    }
  }
}
```

(using the `TypeDesc` fragment from earlier)

Each argument has its own `name`, `description`, and `type`, where the `type` follows the same structure as does the `type` of the field itself. An argument also has a `defaultValue`, if one is part of the schema.

So, for example, the GraphQL demo server's `Query` type has two fields, each with one argument:

```
{
  "name": "Query",
  "fields": [
    {
      "name": "allTrips",
      "args": []
    },
    {
      "name": "getTrip",
      "args": [
        {
          "name": "id",
          "description": "",
          "type": {
            "name": null,
            "kind": "NON_NULL",
            "ofType": {
              "name": "ID",
              "kind": "SCALAR",
              "ofType": null
            }
          },
          "defaultValue": null
        }
      ]
    },
    {
      "name": "findTrips",
      "args": [
        {
          "name": "searchFor",
          "description": "",
          "type": {
            "name": null,
            "kind": "NON_NULL",
            "ofType": {
              "name": "String",
              "kind": "SCALAR",
              "ofType": null
            }
          },
          "defaultValue": null
        }
      ]
```

```
    }
  ]
}
```

Field Deprecation Status

Fields can be marked as "deprecated" in the server's schema. This means that while the field still exists and can be used, it may be removed in the future, and so clients should try to move off of it and onto whatever the replacement is.

When you query on __Type, by default, deprecated fields are *not* returned in the result set. However, the fields field takes an optional includeDeprecated argument, which if you set it to be true, will cause the query to return deprecated fields as well:

```
{
  __schema {
    types {
      name
      fields(includeDeprecated: true) {
        name
        args {
          name
          description
          type {
            ...TypeDesc
          }
          defaultValue
        }
      }
    }
  }
}

fragment TypeDesc on __Type {
  name
  kind
  ofType {
    name
    kind
    ofType {
      name
      kind
      ofType {
        name
        kind
```

```
            }
        }
    }
}
```

However, the results will not tell you anything regarding the deprecation status...
unless you query for that too. You can query for isDeprecated and
deprecationReason for the fields field. isDeprecated will be true or false
depending upon the deprecation state, while deprecationReason will be some sort
of human-readable explanation of why the field was deprecated, or null if there is
no stated reason or if the field is not deprecated.

The GraphQL demo server does not have any deprecated fields (yet). As a result, if
you execute this document:

```
{
  __schema {
    types {
      name
      fields(includeDeprecated: true) {
        name
        args {
          name
          description
          type {
            ...TypeDesc
          }
          defaultValue
        }
        isDeprecated
        deprecationReason
      }
    }
  }
}

fragment TypeDesc on __Type {
  name
  kind
  ofType {
    name
    kind
    ofType {
      name
      kind
      ofType {
```

```
        name
        kind
      }
    }
  }
}
```

...you will see that `isDeprecated` is consistently `false` and `deprecationReason` is consistently `null`.

Type-Specific Results

Everything cited above is common to all types. Some types will have additional information that you can retrieve via introspection queries, such as:

- Objects have `interfaces`, which is either `null` or a list of the type information for the interfaces that this object implements
- Interfaces and unions have `possibleTypes`, listing which types can be members of the interface or union
- Enums can have `enumValues`, which is a list of the possible values for the enum (including deprecation information, as enum values can be deprecated)
- Input objects have `inputFields` as a counterpart to the `fields` available for objects and interfaces
- And so on

Assembling GraphQL

Authentication and the Viewer Pattern

Usually Web services are authenticated, so that only authorized users can access them, whether through a Web app, an Android app, or something else. GraphQL-based Web services are no different in this regard. So far, we have ignored this point, by hitting a public read-only server (the book's demo server) or a local test server (which hopefully nobody but you can access). To be more realistic, we need to consider actual authentication approaches.

GraphQL's schema-based approach, though, adds an interesting implementation wrinkle: different users can see different schemas, based on what they are authorized to do on that server. This would be the equivalent of some REST URLs being eligible for some users but failing with an HTTP 401 error for other users.

Convention Over Specification

Part of what makes GraphQL powerful is a specification, so that everyone can agree on what it means to request and consume GraphQL.

That specification does not span to transports, such as HTTP/HTTPS. And while the GraphQL documentation talks a bit about GraphQL as a Web service, authentication is not one of the topics that it covers, at least as of July 2017.

However, a few conventions have arisen related to authentication in GraphQL. Sometimes, these conventions are simply "ports" of approaches used elsewhere, such as JSON Web Tokens (JWT) as a means of offering authentication. Sometimes, these conventions are more unique to the GraphQL space, or at least are not directly equivalent to techniques that you might use with REST or other counterparts.

199

What You Get Stems From Who You Are

Different GraphQL Web services will use different approaches for authentication. GitHub — whose Web service we will begin exploring in this chapter — uses an API key, known as an authentication token, in an HTTP `Authorization` header. And in the case of GitHub, if you do not have such a header, you get an HTTP 401 response from any GraphQL request.

However, clearly, it is possible for a GraphQL Web service to accept requests without authentication, as the book's demo server and your local copy demonstrate.

What is interesting about GraphQL is how those two concepts can be married together. A GraphQL Web service can have different `Query` and `Mutation` objects, based on the identity (or lack thereof) of whoever is making the GraphQL request.

So, for example:

- An unauthenticated request might have access only to read-only public data, or perhaps can only invoke a mutation that serves as a login, returning a token to be supplied to future GraphQL requests
- An ordinary user might have read-write access to a conventional set of queries and mutations
- An administrator might have additional queries and mutations available, based on the administrative role

One pattern that has emerged in this area is to have an object be available through the `Query` — typically named `viewer` — that represents what the authenticated user has access to. What server-defined type the `viewer` is might vary based upon the user whose authentication credentials were used.

This context-sensitive schema can complicate static GraphQL development, where tools like Apollo-Android want access to the schema for code-generation purposes. If there are unknowable numbers of schemas, based upon uncountable combinations of roles, it may be difficult to figure out how to make Apollo-Android and similar tools happy. For accessing your own server, you may be able to craft some all-powerful account that represents the superset of all possible capabilities, then generate a schema from it.

Exploring GitHub's Approach

Let's take a look at how GitHub approaches authentication and the `viewer` pattern. Most Android developers have at least a little bit of experience with GitHub, considering its current dominance in open source project hosting.

If you have not done so already, you should set up a GitHub account and obtain your personal access token, as is described [earlier in the book](#), as we will be using those items in this chapter.

WARNING: While this chapter and others will focus on reading data from GitHub via its GraphQL Web service, bear in mind that their API is read-write. If you invoke some mutation, you will affect your GitHub account. Be very careful when experimenting with the GitHub API, lest you somehow cause problems with your account.

Using Their GraphiQL

GitHub has their own copy of GraphiQL, known as [the GitHub GraphQL API Explorer](#). In there, enter and execute the following GraphQL document:

```
query whoami {
  viewer {
    login
  }
}
```

Note that you will need to log in, if you have not done so already, via an option towards the upper-right corner of the page.

The output should show your GitHub login name:

```
{
  "data": {
    "viewer": {
      "login": "commonsguy"
    }
  }
}
```

(since your login name is not `commonsguy`, your JSON will have a different value)

Since GitHub only allows authenticated users, you will always have access to a `viewer` object. One of the fields on `viewer` is `login`, which is your GitHub login name.

Using the Token

GitHub's GraphiQL is tightly integrated with the rest of their backend system, and so it is not obvious exactly how they handle the authentication.

However, for their GraphQL endpoint (`https://api.github.com/graphql`), they have documented that your personal access token goes into an HTTP header like:

```
Authorization: bearer ...
```

(where . . . is your token)

Exactly how you arrange for this header to appear will depend on your HTTP client.

For example, [GitHub's documentation](#) shows using `curl` to execute GraphQL from the command line:

```
curl -H "Authorization: bearer ..." https://api.github.com/graphql
```

(where . . . is your token)

GitHub has set up their server such that if you provide no GraphQL document in your HTTP request — as with the above `curl` command — they emit the full schema accessible to you via that token. This is ~1MB, as their API is *huge*, and so we will not reproduce it here.

Of course, it is unlikely that your Android app will use `curl` for its HTTP requests, and so we need to explore how to put a token like this into OkHttp, Apollo-Android, or similar APIs.

Tokens and Android

The [`GitHub/Auth`](#) sample project makes use of the GitHub GraphQL API, through Apollo-Android, to demonstrate how to provide an authentication token to a GraphQL Web service.

Getting the Schema

Any time you use Apollo-Android, you need to get the server schema and have it alongside your GraphQL documents in your source tree. You can obtain the schema using the `apollo-codegen` utility, as we did earlier in the book for the book's demo server.

The sample project already has the GitHub API schema downloaded for you, in the `app/src/main/graphql/com/commonsware/graphql/github/api/` directory.

However, getting the GitHub schema is a bit different than getting the schema for the demo server, as the GitHub server is authenticated. Just as our Apollo-Android code will need to supply the GitHub authentication token, so too does the `apollo-codegen` command. You can do that via the `--header` switch, which takes as its value an HTTP header to add to the request of the GraphQL server.

So, for example, running this command from the `app/` module directory would retrieve the schema and put it in the proper spot:

```
apollo-codegen download-schema https://api.github.com/graphql --header
"Authorization: bearer ..." --output src/main/graphql/com/commonsware/graphql/github/
api/schema.json
```

(where `. . .` is replaced by your authentication token, and where that command should be all on one line but word-wraps here due to the book's line-length limitations)

Adding the Token to the Project

Our Java code will need to do much the same thing, and that means that our Java code will need the GitHub authentication token. However, depending on your project, putting that token into version control might not be a good idea. For example, the author of this book was not interested in putting his authentication token into the book's GitHub repository as part of the `GitHub/Auth` sample project (or other sample projects).

A typical way to manage that is to store the token — or other similar API keys – in a file that you can access from your Gradle build script but is not in version control. In the case of the `GitHub/Auth` sample project, the expectation is that you will put it in `gradle.properties`, such as in the project root directory. Specifically, the `app/build.gradle` file is expecting to encounter a global constant named `GITHUB_TOKEN`

that contains the GitHub authentication token. Adding a `GITHUB_TOKEN=...` line to `gradle.properties` (replacing the ... with your own token) will meet that need.

The Gradle build script will then hoist that `GITHUB_TOKEN` value into the `BuildConfig` class, via `buildConfigField`:

```
apply plugin: 'com.android.application'
apply plugin: 'com.apollographql.android'

dependencies {
  compile 'io.reactivex.rxjava2:rxjava:2.0.2'
  compile 'io.reactivex.rxjava2:rxandroid:2.0.0'
  compile 'com.android.support:recyclerview-v7:26.0.1'
  compile 'com.apollographql.apollo:apollo-rx2-support:0.4.0'
}

android {
    compileSdkVersion 26
    buildToolsVersion "26.0.1"

    defaultConfig {
        minSdkVersion 15
        targetSdkVersion 26
        applicationId "com.commonsware.graphql.github.auth"
        buildConfigField "String", "GITHUB_TOKEN", '"'+GITHUB_TOKEN+'"'

        jackOptions {
            enabled true
        }
    }

  compileOptions {
    sourceCompatibility JavaVersion.VERSION_1_8
    targetCompatibility JavaVersion.VERSION_1_8
  }
}

apollo {
  useSemanticNaming = false
}
```

(from [GitHub/Auth/app/build.gradle](#))

Now, we can use `BuildConfig.GITHUB_TOKEN` to retrieve the GitHub authentication token from Java, without having the actual token appear in version control, if your `.gitignore` excludes `gradle.properties`.

Adding the Token

Injecting an HTTP header into an HTTP request with OkHttp is easy... if you are the one actually making the request. As part of your `Request.Builder` code, you can call `addHeader()` to add an HTTP header. So, if this sample were using OkHttp directly, that is how it would add the `Authorization` header required by GitHub's GraphQL server.

Injecting an HTTP header into an HTTP request with OkHttp is also possible when *something else* is the one making the actual request... such as Apollo-Android. This, however, requires the slightly convoluted approach of an interceptor. OkHttp interceptors are powerful tools for rewriting requests and examining responses for everything that flows through a particular `OkHttpClient` instance.

In the sample project, `ViewerFragment` is our replacement for the `SimpleTripsFragment` shown in some of the preceding book samples. It contains our Apollo-Android code, this time for getting information from GitHub. And therefore it is where our authentication header needs to be added. So, when we set up our `ApolloClient`, we now need a differently-configured `OkHttpClient` than before:

```java
private final OkHttpClient ok=new OkHttpClient.Builder()
  .addInterceptor(chain -> {
    Request orig=chain.request();
    Request.Builder builder=orig.newBuilder()
      .method(orig.method(), orig.body())
      .header("Authorization", "bearer "+BuildConfig.GITHUB_TOKEN);

    return(chain.proceed(builder.build()));
  })
  .build();
private ApolloClient apolloClient=ApolloClient.builder()
  .okHttpClient(ok)
  .serverUrl("https://api.github.com/graphql")
  .build();
```

(from GitHub/Auth/app/src/main/java/com/commonsware/graphql/github/ViewerFragment.java)

Here, we use a lambda expression to add an interceptor to OkHttp. We get the original `Request` that was going to be made, create a new `Request.Builder` from that `Request`, and use `header()` to add our `Authorization` header with our `GITHUB_TOKEN`. The interceptor then proceeds, using the new `Request` built from the `Request.Builder`. The net effect is that we have augmented the `Request` with the `Authorization` header.

The Busy Coder's Guide to Android Development has more material on OkHttp interceptors, if you are interested in more details about using this technique.

Making the Request

From there, the rest of the code is reminiscent of what we had before:

```java
package com.commonsware.graphql.github;

import android.annotation.SuppressLint;
import android.os.Bundle;
import android.support.v7.widget.DividerItemDecoration;
import android.support.v7.widget.LinearLayoutManager;
import android.util.Log;
import android.view.View;
import android.widget.Toast;
import com.apollographql.apollo.ApolloClient;
import com.apollographql.apollo.api.Response;
import com.apollographql.apollo.rx2.Rx2Apollo;
import com.commonsware.graphql.github.api.Whoami;
import io.reactivex.Observable;
import io.reactivex.android.schedulers.AndroidSchedulers;
import io.reactivex.disposables.Disposable;
import io.reactivex.schedulers.Schedulers;
import okhttp3.OkHttpClient;
import okhttp3.Request;

public class ViewerFragment extends RecyclerViewFragment {
  private final OkHttpClient ok=new OkHttpClient.Builder()
    .addInterceptor(chain -> {
      Request orig=chain.request();
      Request.Builder builder=orig.newBuilder()
        .method(orig.method(), orig.body())
        .header("Authorization", "bearer "+BuildConfig.GITHUB_TOKEN);

      return(chain.proceed(builder.build()));
    })
    .build();
  private ApolloClient apolloClient=ApolloClient.builder()
    .okHttpClient(ok)
    .serverUrl("https://api.github.com/graphql")
    .build();
  private Observable<Whoami.Data> observable;
  private Disposable sub;

  @Override
```

206

```java
public void onCreate(Bundle savedInstanceState) {
  super.onCreate(savedInstanceState);

  setRetainInstance(true);

  observable=Rx2Apollo.from(apolloClient.query(new Whoami()).watcher())
    .subscribeOn(Schedulers.io())
    .map(response -> (getFields(response)))
    .cache()
    .observeOn(AndroidSchedulers.mainThread());
}

@Override
public void onViewCreated(View v, Bundle savedInstanceState) {
  super.onViewCreated(v, savedInstanceState);

  setLayoutManager(new LinearLayoutManager(getActivity()));

  getRecyclerView()
    .addItemDecoration(new DividerItemDecoration(getActivity(),
      LinearLayoutManager.VERTICAL));

  unsub();
  sub=observable.subscribe(
    s -> setLogin(s.viewer().login()),
    error -> {
      Toast
        .makeText(getActivity(), error.getMessage(), Toast.LENGTH_LONG)
        .show();
      Log.e(getClass().getSimpleName(), "Exception processing request",
        error);
    });
}

@Override
public void onDestroy() {
  unsub();

  super.onDestroy();
}

private void unsub() {
  if (sub!=null && !sub.isDisposed()) {
    sub.dispose();
  }
}

private Whoami.Data getFields(Response<Whoami.Data> response) {
```

207

```java
    if (response.hasErrors()) {
      throw new RuntimeException(response.errors().get(0).message());
    }

    return(response.data());
  }

  private void setLogin(String login) {
    ((MainActivity)getActivity()).setLogin(login);
  }
}
```

<div align="center">(from <u>GitHub/Auth/app/src/main/java/com/commonsware/graphql/github/ViewerFragment.java</u>)</div>

Our fragment sets up the `ApolloClient` with the augmented `OkHttpClient` and the GitHub API URL (`https://api.github.com/graphql`). We now have a `Whoami` query that we can run, courtesy of a GraphQL document in our source tree:

```graphql
query whoami {
  viewer {
    login
  }
}
```

<div align="center">(from <u>GitHub/Auth/app/src/main/graphql/com/commonsware/graphql/github/api/GitHubApi.graphql</u>)</div>

This `whoami` query gets the `viewer` and just retrieves the `login` name of the authenticated user.

From there, the Java code winds through RxJava/RxAndroid to make the GraphQL request, eventually resulting in a call to `setLogin()` on the `MainActivity`. That, in turn, updates the title in the action bar:

```java
package com.commonsware.graphql.github;

import android.app.Activity;
import android.os.Bundle;

public class MainActivity extends Activity {
  @Override
  public void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);

    if (getFragmentManager().findFragmentById(android.R.id.content) == null) {
      getFragmentManager().beginTransaction()
                          .add(android.R.id.content,
                              new ViewerFragment()).commit();
```

```
    }
  }

  void setLogin(String login) {
    setTitle("GitHub: "+login);
  }
}
```

(from GitHub/Auth/app/src/main/java/com/commonsware/graphql/github/MainActivity.java)

The net result — assuming that you have a valid GitHub authentication token set up as described — is that when you run the app, your GitHub login name appears in the title bar of an otherwise empty activity:
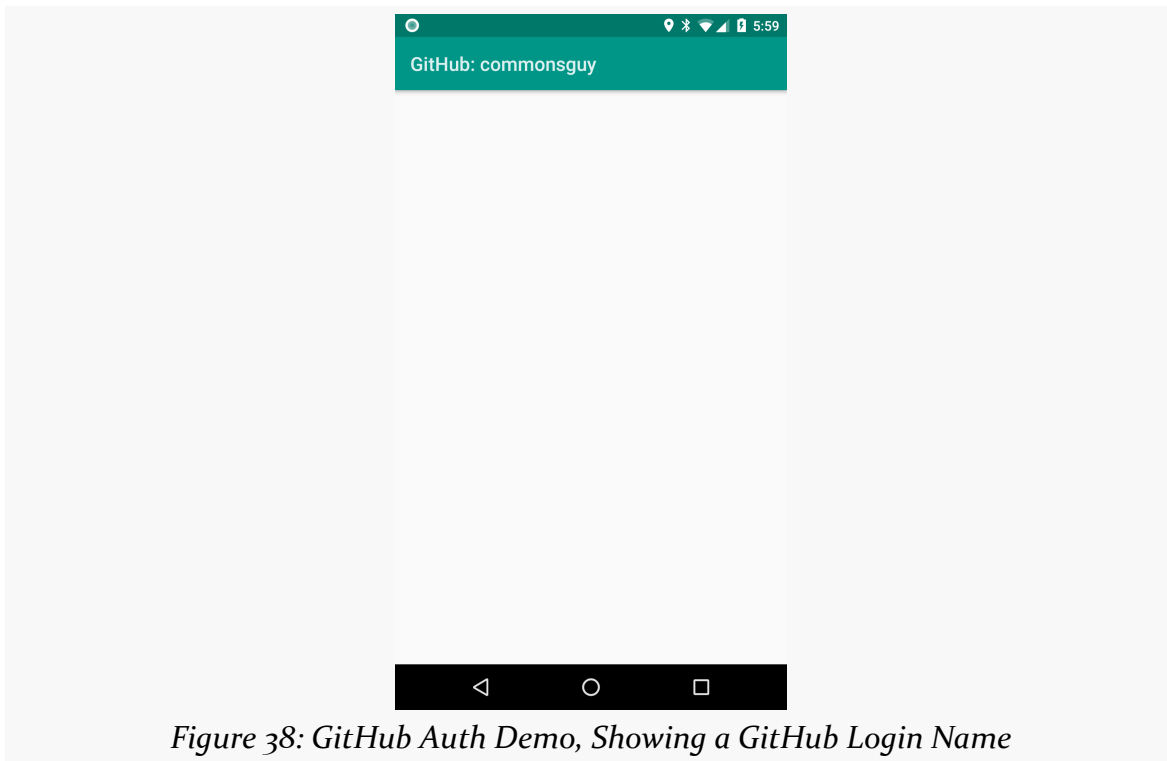

Figure 38: GitHub Auth Demo, Showing a GitHub Login Name

Queries and Paging

Sometimes, there is just too much data.

Users of Web browsers are used to the notion of data paging, whether that comes in the form of pages of search engine results, "append as you scroll" patterns for Twitter timelines, or in other forms.

For data sets known to be potentially large, Web service APIs often offer paging, where clients can indicate a starting point and a number of items to retrieve from the Web service. GraphQL-powered Web services are no exception. In particular, there is one convention for implementing such paging, promoted by the Relay project for Web clients. Since having a Relay-compatible GraphQL server is often an objective, the Relay paging convention has become fairly popular.

In this chapter, we will explore this convention, what it looks like from a GraphQL standpoint, and how to consume this sort of page-at-a-time results from an Apollo-Android client.

Taking a Page from GitHub

GitHub's GraphQL API makes use of paged queries quite a bit, in part because GitHub has geared its API to handle arbitrarily complex scenarios, including huge projects with lots of contributors and tons of code.

So, let's start by invoking a GitHub paged query to get the first page of results.

Starring Some Repositories

The specific query that we are going to invoke retrieves the `starredRepositories`

211

field for the `viewer`. As the name suggests, `starredRepositories` represents repositories starred by the logged-in user.

To make this query be effective — both for interactive exploration and for use with this chapter's sample app — it will help if you have a bunch of starred repositories. Ideally, you have 11 or more.

You can see your starred repositories from your GitHub profile page, in the "Stars" tab:
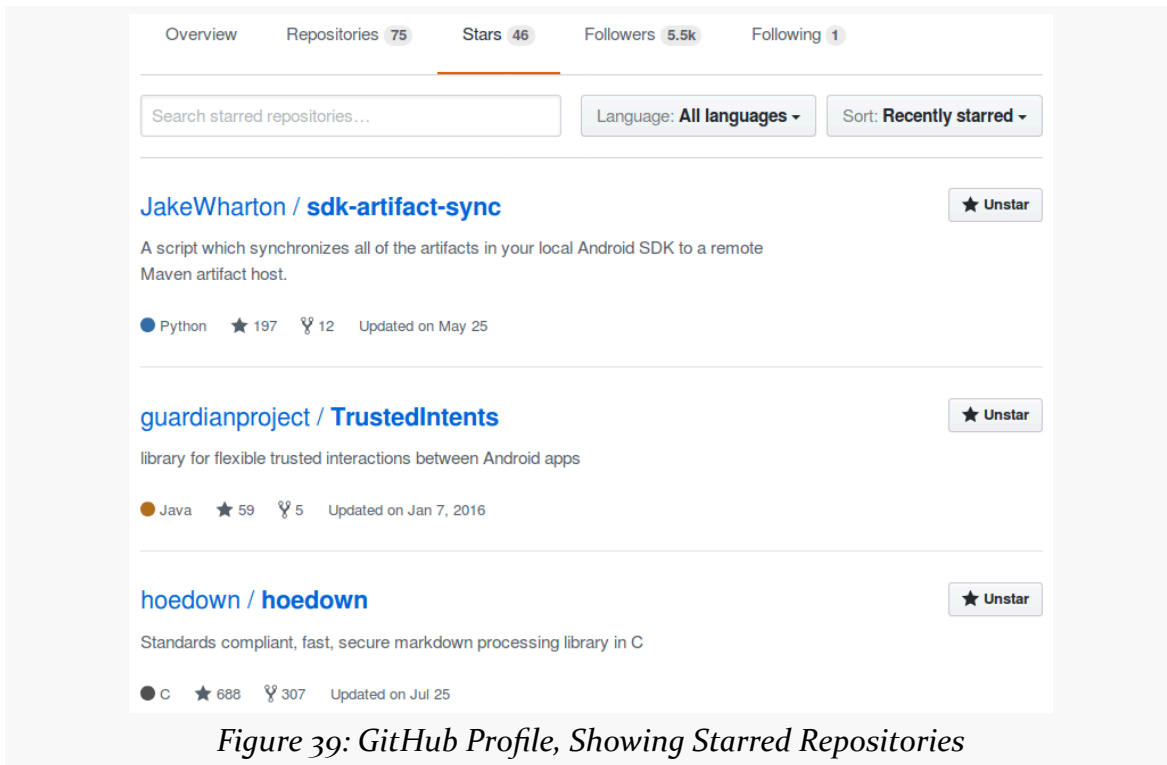

Figure 39: GitHub Profile, Showing Starred Repositories

To star a repository, on any GitHub page for that repository, find the button bar towards the top of any GitHub repo page:
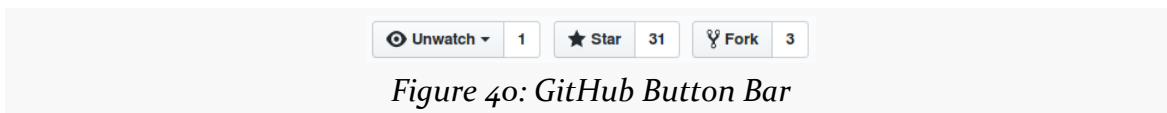

Figure 40: GitHub Button Bar

Click the "Star" button to star it, at which point the button's caption will change to "Unstar".

212

Getting a Page of Results

Once you have some starred repositories, visit [the GitHub GraphQL API Explorer](#) and fill in the following GraphQL document in the GraphiQL pane:

```
query myStars($first: Int, $after: String) {
  viewer {
    login
    starredRepositories(first: $first, after: $after) {
      edges {
        cursor
        node {
          id
          name
        }
      }
      pageInfo {
        hasNextPage
      }
    }
  }
}
```

Then, in the Query Variables pane, fill in:

```
{
  "first": 5
}
```

(we can skip the `after` variable, as it will default to `null`, which is what we want here for now)

Then, execute the query using the GraphiQL run button. This should give you a JSON output showing 5 of your starred repositories:

```
{
  "data": {
    "viewer": {
      "login": "commonsguy",
      "starredRepositories": {
        "edges": [
          {
            "cursor": "Y3Vyc29yOnYyOpHOAC50Bg==",
            "node": {
              "id": "MDEwOlJlcG9zaXRvcnkxNzc2MjA=",
```

```
          "name": "em-proxy"
        }
      },
      {
        "cursor": "Y3Vyc29yOnYyOpHOAC5OBw==",
        "node": {
          "id": "MDEwOlJlcG9zaXRvcnkyNTU5MjU=",
          "name": "cwac-task"
        }
      },
      {
        "cursor": "Y3Vyc29yOnYyOpHOAC5OCA==",
        "node": {
          "id": "MDEwOlJlcG9zaXRvcnkyNTU5NDI=",
          "name": "cwac-bus"
        }
      },
      {
        "cursor": "Y3Vyc29yOnYyOpHOAC5OCQ==",
        "node": {
          "id": "MDEwOlJlcG9zaXRvcnkyNTU5NTA=",
          "name": "cwac-cache"
        }
      },
      {
        "cursor": "Y3Vyc29yOnYyOpHOAC5OCg==",
        "node": {
          "id": "MDEwOlJlcG9zaXRvcnkyNTU5ODE=",
          "name": "cwac-custmenu"
        }
      }
    ],
    "pageInfo": {
      "hasNextPage": true
    }
  }
}
}
```

Examining the Query and Its Results

Given that we can get our first page's worth of starred repositories, let's break down
the GraphQL query and the results that we got back.

The Viewer

As outlined in the previous chapter, GitHub uses the viewer pattern as part of its authentication and authorization approach. Your GitHub API key grants you access to whatever the viewer field offers to you. In the previous chapter's samples, we just retrieved the login field from viewer. Here, we are also requesting the starredRepositories field.

The starredRepositories

GitHub's starredRepositories field follows the Relay Cursor Connections Specification.

Specifically starredRepositories is a list of StarredRepositoryConnection objects, fulfilling one of the Relay specification's requirements: paged queries return objects whose type names end in Connection.

Typically, such fields will have arguments that indicate what "page" of results should be returned. Sometimes, the GraphQL endpoint will require such fields. GitHub's does not — if you remove the variables from your query and run it again, you should get back all of your starred repositories, not just 5.

However, originally, we used one of those arguments: first, indicating how many items we want back in the results. Why this is named first and not something like pageSize or count will become a bit clearer as we proceed.

The Edges

Relay-style Connection types have two required fields. The first of these is edges, and it must return what the Relay specification refers to as an "edge type". That is some object with its own pair of required fields: cursor and node.

cursor is an opaque string... at least in terms of how it is represented in the JSON (and therefore in the client-side code). The server is welcome to define it as something else (e.g., a scalar), but clients can safely assume that it will be a string. Here, "opaque" means "do not make any assumptions about the nature of its value". We will see shortly where and how we use this value.

The node returns the actual data associated with this entry. In the case of starredRepositories, a node is a Repository object, which has lots of fields for

215

details about that GitHub repository. We happen to be retrieving the repository's `id` and `name`.

An edge can contain additional fields beyond these two. In the case of `starredRepositories`, the edge also has a `starredAt` field, indicating the `DateTime` when you starred that repository.

The Page Info

The `pageInfo` field will be an object with metadata about this page's worth of results. The Relay specification requires it to have at least two fields: `hasPreviousPage` and `hasNextPage`. As one might expect, these are `boolean` values, indicating whether we can move forwards or backwards from the current page.

However, only one of the two will be relevant for any given query. In our case, we are using `first` to indicate how many starred repositories we want, and according to the Relay specification, only `hasNextPage` needs to hold a meaningful value. `hasPreviousPage` will be defined but will always be `false`.

In our case, we requested fewer starred repositories (5) that existed, and so the result for `hasNextPage` is `true`.

We will see both of these fields in use in the next section.

Please, GitHub, Can I Have Some More?

The Relay specification supports both forwards and backwards paging, by way of four arguments (in this case, on `starredRepositories`). They are organized into two pairs:

- `first` and `after`
- `last` and `before`

`first` is for forward paging. The `first` argument indicates how many results you want (maximum). `after` is a `cursor` value from an edge; the returned edges should be the first ones after the `after` edge.

For example, examine your JSON output from the query and find the fourth edge's `cursor` value. Then, use that in the variables JSON for `after`:

```
{
  "first": 5,
  "after": "Y3Vyc29yOnYyOpHOAC50CQ=="
}
```

Then, run that query with the revised variables. You should get five repositories in the output, starting with the fifth one from your original run:

```
{
  "data": {
    "viewer": {
      "login": "commonsguy",
      "starredRepositories": {
        "edges": [
          {
            "cursor": "Y3Vyc29yOnYyOpHOAC50Cg==",
            "node": {
              "id": "MDEwOlJlcG9zaXRvcnkyNTU5ODE=",
              "name": "cwac-custmenu"
            }
          },
          {
            "cursor": "Y3Vyc29yOnYyOpHOAC50Cw==",
            "node": {
              "id": "MDEwOlJlcG9zaXRvcnkyNTYwODE=",
              "name": "vidtry"
            }
          },
          {
            "cursor": "Y3Vyc29yOnYyOpHOAC50DA==",
            "node": {
              "id": "MDEwOlJlcG9zaXRvcnkyNTYyMjk=",
              "name": "cwac-sacklist"
            }
          },
          {
            "cursor": "Y3Vyc29yOnYyOpHOAC50DQ==",
            "node": {
              "id": "MDEwOlJlcG9zaXRvcnkyNTYyNDU=",
              "name": "cwac-endless"
            }
          },
          {
            "cursor": "Y3Vyc29yOnYyOpHOAC50Dg==",
            "node": {
              "id": "MDEwOlJlcG9zaXRvcnkyNTc3NzI=",
              "name": "cwac-merge"
```

```
        }
      }
    ],
    "pageInfo": {
      "hasNextPage": true
    }
  }
}
}
}
```

Normally — particularly from a client app — you would use the last `cursor` value, rather than overlapping as we did here. That was for illustration purposes, so you see that the returned values start with the edges after the supplied `after` value.

`last` and `before` work similarly. `last` indicates how many you want, while `before` provides the `cursor` value that the results should precede. And, in this case, you should retrieve `hasPreviousPage` instead of `hasNextPage` in the `pageInfo`, to determine if there is yet more data to be retrieved.

If you leave off `before` or `after` — as we did in the original `first` query — the GraphQL endpoint should give you the ends of the results (e.g., the very first items if you use `first` without `after`).

So, the algorithm to get all of the pages, going in the forward direction, looks like this bit of pseudo-code:

```
cursor=null

do
  query for first N items after cursor
  cursor=value from last item
while no errors and hasNextPage
```

Paging with Apollo-Android

Translating that pseudo-code into Java in an Android app is a bit tricky. Apollo-Android is not a problem — we just supply our values via the query builder. The problem is in coming up with an RxJava approach that handles this sort of flow.

The [GitHub/Starred](#) sample project illustrates this.

218

Defining the GraphQL

We use a slightly different GraphQL document, one that takes advantage of another starredRepositories argument: orderBy:

```
query myStars($first: Int, $after: String) {
  viewer {
    login
    starredRepositories(first: $first, after: $after, orderBy: {field: STARRED_AT, direction: DESC}) {
      edges {
        cursor
        node {
          id
          name
        }
      }
      pageInfo {
        hasNextPage
      }
    }
  }
}
```

(from GitHub/Starred/app/src/main/graphql/com/commonsware/graphql/github/api/GitHubApi.graphql)

As the name suggests, orderBy controls the sort order of the results. It takes an input object with two fields:

- field, indicating what field we want to sort on, where in the current edition of the GraphQL API the only option is STARRED_AT
- direction, which will be ASC for ascending and DESC for descending

Rather than make this configurable, we are hard-coding in the document that we want to order the results by the date the repository was starred at, descending.

Otherwise, this document is the same one that we were using in the GitHub GraphQL Explorer earlier in this chapter.

RxJava Recipe

For the purposes of this sample, we are going to retrieve all pages of the result up front, when the activity and ViewerFragment are displayed. Occasionally, this is the right answer. More often, you will load pages based on user input, so that you do not unnecessarily consume data and memory for content that the user will never see.

From an Rx standpoint, what would be great is a way to have a single Observable chain that we initialize at the outset, one that retrieves all of the starred repositories.

The simple way to do that would be to request all of the repositories in one shot. GitHub seems to support this, by fetching `starredRepositories` with neither `first` nor `last`. However, that defeats the point of the sample, which is to show paging.

Making the paging that much more complex is that the second page request requires data from the first page — specifically, a `cursor` value. The third page request requires a `cursor` from the second page, and so on. This is in contrast to Web services that use response-independent identifiers of pages (e.g., a page number and a page size). While RxJava has a fair bit of functionality for making an `Observable` chain out of other `Observable` objects, that gets rather complicated when the later `Observable` objects depend on the earlier ones. Some solutions involve recursion, which can get you in trouble for large result sets.

Instead, in this sample app, we take advantage of the fact that Apollo-Android does not *require* RxJava, or even asynchronous operation. The result of calling `query()` or `mutation()` on an `ApolloClient` is an implementation of an `ApolloCall`. Its native API is reminiscent of OkHttp, with `execute()` and `enqueue()` methods. `execute()` works synchronously. So, we can create a custom `Observable` that takes advantage of that synchronous operation to be able to get the results for each page based upon the previous page. RxJava can arrange to do all of that work on its own background thread, courtesy of `subscribeOn()`.

So, we start with a synchronous `getPage()` method that returns the result of a GraphQL query for a particular page's worth of results:

```java
private Response<MyStars.Data> getPage(String cursor) throws ApolloException {
  return(apolloClient.query(MyStars.builder().first(5).after(cursor).build()).execute());
}
```

(from GitHub/Starred/app/src/main/java/com/commonsware/graphql/github/ViewerFragment.java)

Here, we use `query()` with an `ApolloClient` as in previous examples, but we `execute()` it directly. This is a blocking call, performing the network I/O and parsing, returning the `Response`. We use the supplied `cursor`, along with a page size of 5 items. That is a very small page size, mostly to help you see the chained responses. In a production app, choose a somewhat larger page size.

Wrapping that in an RxJava `Observable` that handles all of the pages is handled by a `getPages()` method, based upon [this Stack Overflow answer](#):

```java
// based on https://stackoverflow.com/a/45938570/115145

private Observable<Response<MyStars.Data>> getPages() {
  return(Observable.generate(() -> (getPage(null)),
```

```
    (previousPage, responseEmitter) -> {
      MyStars.StarredRepositories repos=previousPage.data().viewer().starredRepositories();
      List<MyStars.Edge> edges=repos.edges();
      MyStars.Edge last=edges.get(edges.size()-1);
      Response<MyStars.Data> result=getPage(last.cursor());

      responseEmitter.onNext(result);

      if (result.hasErrors() || !repos.pageInfo().hasNextPage()) {
        responseEmitter.onComplete();
      }

      return(result);
    }));
}
```

(from GitHub/Starred/app/src/main/java/com/commonsware/graphql/github/ViewerFragment.java)

We use the generate() method on Observable to... well... generate an Observable. generate() takes two callback objects (or, in this case, lambda expressions).

The first lambda expression generates the initial state. Here, we call getPage(null), to retrieve the first Response, using a null value for the cursor.

The second lambda expression is called for each Response that we generate, staring with the one from the initial state. We are given that Response, along with an Emitter, which we use to emit the next response and complete the Observable when we are out of data. Here, we:

- Get the starredRepositories field from the results of the previous response
- Get the edges field from the starredRepositories
- Find the last edge from the edges
- Call getPage() with the cursor value from that last edge
- Emit that new Response from the getPage() call

At that point, if we are done — we received an error or hasNextPage is false — we call onComplete() on the Emitter, to indicate that there is no more data to be retrieved and that this lambda expression does not need to be invoked again.

Then, up on onCreate() of ViewerFragment, we set up our Observable based upon getPages():

```
  @Override
  public void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);

    setRetainInstance(true);
```

221

```
observable=getPages()
  .subscribeOn(Schedulers.io())
  .map(response -> (getFields(response)))
  .cache()
  .observeOn(AndroidSchedulers.mainThread());
}
```

Along the way, we map() the Response objects into MyStars.Data objects or exceptions, using a getFields() similar to the implementations found in earlier samples:

```
private MyStars.Data getFields(Response<MyStars.Data> response) {
  if (response.hasErrors()) {
    throw new RuntimeException(response.errors().get(0).message());
  }

  return(response.data());
}
```

The Rest of the RecyclerView

By and large, the rest of the sample app follows the RecyclerView-based samples seen earlier in the book… with some wrinkles.

Our RecyclerView.ViewHolder is once again called RowHolder. It maps the name field from our GraphQL response into a simple TextView in the row:

```
private static class RowHolder extends RecyclerView.ViewHolder {
  final private TextView name;

  RowHolder(View itemView) {
    super(itemView);

    name=(TextView)itemView.findViewById(android.R.id.text1);
  }

  void bind(MyStars.Node node) {
    name.setText(node.name());
  }
}
```

Our `RecyclerView.Adapter` is now called `RepoAdapter`:

```java
private static class RepoAdapter extends RecyclerView.Adapter<RowHolder> {
  private final LayoutInflater inflater;
  private final List<MyStars.Edge> edges=new ArrayList<>();

  RepoAdapter(LayoutInflater inflater) {
    this.inflater=inflater;
  }

  @Override
  public RowHolder onCreateViewHolder(ViewGroup parent,
                                      int viewType) {
    return(new RowHolder(inflater
      .inflate(android.R.layout.simple_list_item_1, parent, false)));
  }

  @Override
  public void onBindViewHolder(RowHolder holder, int position) {
    holder.bind(edges.get(position).node());
  }

  @Override
  public int getItemCount() {
    return(edges.size());
  }

  private void addEdges(List<MyStars.Edge> moarEdges) {
    int size=edges.size();

    edges.addAll(moarEdges);
    notifyItemRangeInserted(size, moarEdges.size());
  }
}
```

`RepoAdapter` has to take into account that we are getting our starred repository data a page at a time. We could set up the RxJava `Observable` chain to concatenate all of those pages into a single `ArrayList` and use that. But then we cannot show anything until all the results are in. Instead, we incrementally update the `RecyclerView` contents based upon whatever repositories we have received to date. So, `RepoAdapter` has an `addEdges()` method, that takes `MyStars.Edge` objects (modeling the edge of our GraphQL response) and adds them to its own master roster of edges.

addEdges() also calls notifyItemRangeInserted(), to tell the RecyclerView that we added more rows to the list.

Our RepoAdapter instance is held in a repoMan field, initialized in onViewCreated() of our ViewerFragment:

```java
@Override
public void onViewCreated(View v, Bundle savedInstanceState) {
  super.onViewCreated(v, savedInstanceState);

  setLayoutManager(new LinearLayoutManager(getActivity()));

  getRecyclerView()
    .addItemDecoration(new DividerItemDecoration(getActivity(),
      LinearLayoutManager.VERTICAL));
  repoMan=new RepoAdapter(getActivity().getLayoutInflater());
  setAdapter(repoMan);

  unsub();
  sub=observable.subscribe(
    s -> applyResults(s.viewer()),
    error -> {
      Toast
        .makeText(getActivity(), error.getMessage(), Toast.LENGTH_LONG)
        .show();
      Log.e(getClass().getSimpleName(), "Exception processing request",
        error);
    });
}
```

(from GitHub/Starred/app/src/main/java/com/commonsware/graphql/github/ViewerFragment.java)

There, we also subscribe() to our Observable and begin consuming the GraphQL page responses. Along the way, we use applyResults() both to update the title bar (as in the sample from [the chapter on authentication](#)) and call the addEdges() method on the RepoAdapter:

```java
private void applyResults(MyStars.Viewer viewer) {
  ((MainActivity)getActivity()).setLogin(viewer.login());
  repoMan.addEdges(viewer.starredRepositories().edges());
}
```

(from GitHub/Starred/app/src/main/java/com/commonsware/graphql/github/ViewerFragment.java)

The result is that our RecyclerView is populated with the starred repositories, in batches of five:

224

Using GraphQL in Android

Securing GraphQL

Any time we deal with clients and servers, security should have paramount importance. GraphQL does not cause security issues to magically vanish, any more than using REST does. So, as we develop our Android GraphQL clients, we need to consider what we need to do to make sure that our users are protected, with particular emphasis on what we can do on the client side to help with that protection.

Securing the Network

Our principal concern with GraphQL is ensuring that the communications meet the "CIA" objectives:

- Confidentiality: others cannot read the communications
- Integrity: others cannot modify the communications
- Availability: others cannot block the communications

By and large, this involves SSL/TLS, the same as with any other Web service. While there is nothing specific to GraphQL about using HTTPS, it is important to review the steps, as (unfortunately) not all of these are ingrained in every developer's thought process while programming.

This chapter provides a "quick and dirty" explanation of various techniques. See *The Busy Coder's Guide to Android Development* for in-depth coverage of these topics.

SSL Basics

Hopefully, the GraphQL server that you are talking to has set up an SSL certificate and is using that to encrypt the communications between your client and the server.

227

Courtesy of services like [Let's Encrypt](#), SSL certificates can be free of charge, and so certificate cost should not be a concern for the server team.

That will allow you to use an `https` URL with your GraphQL client, whether that is using OkHttp directly, using Apollo-Android, or perhaps using something else.

However, we still have some security issues:

- An attacker could claim to be the GraphQL server, when in reality it is some sort of middleman proxy. Your code thinks it is talking directly to the GraphQL server, but instead it is talking to the proxy, and the proxy is separately forwarding your requests to the GraphQL server. This allows the proxy to see everything. This is referred to as a MITM attack, where MITM is "man in the middle" (or, possibly, "Martian in the middle"). Ideally, your app would detect this scenario and refuse to connect to the attacker's proxy.
- While the OkHttp and Apollo-Android APIs make it easy to try to use HTTPS for all requests, you might screw up and accidentally use HTTP. If the server supports both HTTP and HTTPS, your requests would go over HTTP without you realizing it. Ideally, your code would prevent such accidental unencrypted communications.

OkHttp Certificate Pinning

One way to address MITM attacks is through certificate pinning. You include in your app details about the specific SSL certificate that you expect to see when connecting to a particular server. An attacker will not have that certificate — instead, they will use a fraudulent one. When your app connects to the attacker's proxy, it can determine that the attacker is using an unexpected certificate and refuse to connect.

OkHttp offers an implementation of certificate pinning that you can use. The `Trips/CW/StaticOkPin` sample project demonstrates its use.

Establishing a Pin

When you create your `OkHttpClient` instance, use an `OkHttpClient.Builder`. Then, on the `Builder`, you can call `certificatePinner()`, supplying a `CertificatePinner` containing your pinning rules. You create a `CertificatePinner` via a `CertificatePinner.Builder`, where you can `add()` particular pins:

```
private OkHttpClient ok=new OkHttpClient.Builder()
  .certificatePinner(new CertificatePinner.Builder()
```

228

```
        .add("graphql-demo.commonsware.com", "sha256/3Zeb2W9lSpG9DrsLH03DRCxcu0j7BFyLVXcR7cZW9tQ=")
        .build())
  .build();
```

The add() method takes two parameters:

- The hostname of the Web service
- A value used to validate the certificate that OkHttp receives from that Web service

While we refer to this as "certificate pinning", in truth, this is "public key pinning", as the validation value (the "pin") is a hash of the public key information from the certificate itself.

To generate that value, you will need to use a tool like openssl. Given a PEM file named server.crt, you can generate the hash for that server using the following command:

```
openssl x509 -in server.crt -pubkey -noout | openssl pkey -pubin -outform der |
openssl dgst -sha256 -binary | openssl enc -base64
```

(NOTE: this should appear all one one line but will be word-wrapped to the size of the book page)

And, since this is generating a SHA-256 hash, the actual OkHttp pin will be sha256/, followed by the value generated by the aforementioned openssl snippet.

Given that you have an OkHttpClient configured with the pin, you can then use it directly, via Apollo-Android, etc. Testing to confirm that the pinning is working is merely a matter of changing the pin value (e.g., in the sample app, change the leading 3 to a 4) and confirming that you can no longer connect to the GraphQL server due to a pin mismatch.

Implementing a Pin Set

However, bear in mind that when you use pins, when the server changes SSL certificates, the app needs to change to the new pins.

If you make multiple add() calls on the CertificatePinner.Builder for the same hostname, they represent a logical OR. If the certificate for a given connection matches either of those pins, the connection is accepted.

229

Hence, several months before your existing SSL certificate expires, you can obtain a new one, then add a pin for it to the Java code. Have the server switch to the new certificate before the old one expires. Everybody who updated their app after you published the update with the second pin will have no problems, as their app will work with both the old and the new certificates.

Pinning Against a Root CA

The problem with having two pins is that not everyone will update to the newer app version in time. You might consider this to be a feature, not a bug, and use the specific exception to tell the user that they need to update their app before they continue using it. In other cases, preventing the user from using the app, just because the SSL certificate changed, might be considered inappropriate.

Another approach is to pin not against your server's specific SSL certificate, but instead pin against the root SSL certificate, from the certificate authority (CA) that you obtained your SSL certificate from. So long as any replacement certificates come from the same CA, your pins do not need to change, and so apps depending on those pins continue to work. However:

- If you need to change CAs for some reason, this will not help, and you will need to implement a pin set
- An attacker who fraudulently obtains a certificate from your CA will be able to successfully intercept your communications, since your pin will match their certificate as well as yours (and any others coming from that CA)

The simplest way to find the pin values for a root CA is to intentionally crash. Use an invalid pin in your `OkHttpClient.Builder` configuration, such as `sha256/AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA=`. Then, when you crash with the connection failure, look in LogCat for a message from OkHttp akin to the following:

```
 Peer certificate chain:
   sha256/3Zeb2W9lSpG9DrsLHO3DRCxcuOj7BFyLVXcR7cZW9tQ=: CN=graphql-
demo.commonsware.com,OU=PositiveSSL,OU=Domain Control Validated
   sha256/klO23nT2ehFDXCfx3eHTDRESMz3asj1muO+4aIdjiuY=: CN=COMODO RSA Domain
Validation Secure Server CA,O=COMODO CA Limited,L=Salford,ST=Greater Manchester,C=GB
   sha256/grX4Ta9HpZx6tSHkmCrvpApTQGo67CYDnvprLg5yRME=: CN=COMODO RSA Certification
Authority,O=COMODO CA Limited,L=Salford,ST=Greater Manchester,C=GB
```

The first pin value in this chain will be the one from your specific server and should match what you generated via `openssl`. The other pin values represent the rest of the SSL certificate chain, from your server to the root CA. Using the last pin value

will pin you to the root CA, instead of for your server.

OkHttp Cleartext Ban

Strictly speaking, OkHttp does not have a means of blocking cleartext traffic. However, Jesse Wilson demonstrates the equivalent solution, which is to limit the types of connections that an `OkHttpClient` will accept to those that imply encryption:

```
private OkHttpClient ok=new OkHttpClient.Builder()
    .connectionSpecs(Arrays.asList(
        ConnectionSpec.MODERN_TLS,
        ConnectionSpec.COMPATIBLE_TLS))
    .build();
```

Network Security Configuration

Android 7.0 added a "network security configuration" subsystem. This allows you to define HTTPS requirements, such as certificate pins, in an XML resource. These are then applied automatically to all Java-based socket communications. As of Android 8.0, the cleartext traffic ban requested via a network security configuration also affects `WebView`. Hence, if you want to set up common rules that might transcend the use of a single `OkHttpClient`, network security configuration is one way to go about it.

The `Trips/CW/StaticSecure` sample project contains a `res/xml/network_security.xml` configuration file that bans cleartext traffic to `graphql-demo.commonsware.com` and sets up a certificate pin, akin to what we did using `OkHttpClient.Builder`:

```xml
<?xml version="1.0" encoding="utf-8"?>
<network-security-config>
  <domain-config cleartextTrafficPermitted="false">
    <domain includeSubdomains="false">graphql-demo.commonsware.com</domain>
    <pin-set expiration="2020-01-01">
      <pin digest="SHA-256">3Zeb2W9lSpG9DrsLH03DRCxcuOj7BFyLVXcR7cZW9tQ=</pin>
    </pin-set>
  </domain-config>
</network-security-config>
```

(from Trips/CW/StaticSecure/app/src/main/res/xml/network_security.xml)

231

The `<domain>` element inside the `<domain-config]` element indicates that the `<domain-config>` is defining rules for accessing a specific domain (`graphql-demo.commonsware.com` in this case). The `cleartextTrafficPermitted="false"` attribute bans cleartext traffic to this domain, and the `<pin>` inside of the `<pin-set>` uses the same SHA-256 hash as we used with `OkHttpClient.Builder`.

The `<pin-set>` also contains an `expiration` attribute. When this date arrives, *the pins in this set will be ignored*, and traffic using certificates that does not match the pin will still be allowed. This weakens security, as MITM attacks will be possible as of the indicated date. However, by "failing open", it allows the app to continue working, even though the server might have switched to a new certificate. Skipping `expiration` will result in pins that work indefinitely.

This particular pin pins the server's own certificate. As with OkHttp, network security configuration will accept a connection if any certificate in its chain matches any of the pins. So, you can use a pin for the root CA's certificate, if you wish.

To teach Android 7.0+ about these rules, add an `android:networkSecurityConfig` attribute to the `<application>` element in the manifest:

```
<application
  android:icon="@drawable/ic_launcher"
  android:label="@string/app_name"
  android:networkSecurityConfig="@xml/network_security"
  android:theme="@style/Theme.Apptheme">
```

(from [Trips/CW/StaticSecure/app/src/main/AndroidManifest.xml](#))

No additional configuration is required.

CWAC-NetSecurity

The problem with network security configuration is that it is only available on Android 7.0+. If you like the feature set of network security configuration but need to support older devices, the author of this book maintains [CWAC-NetSecurity](#), a backport of network security configuration that works on API Level 17+ (Android 4.2+). CWAC-NetSecurity also offers additional capabilities, such as on-the-fly "soft" certificate pins.

However, CWAC-NetSecurity does not work automatically, the way the native Android 7.0+ network security configuration does. There are a few additional manual steps, illustrated in the same [Trips/CW/StaticSecure](#) sample project:

- Add the CWAC Maven artifact repository as a source of artifacts, such as via the `allprojects` closure in the top-level `build.gradle` file:

```
buildscript {
    repositories {
        jcenter()
    }
    dependencies {
        classpath 'com.android.tools.build:gradle:2.3.3'
        classpath 'com.apollographql.apollo:gradle-plugin:0.4.0'
    }
}

allprojects {
    repositories {
        jcenter()
        maven { url "https://s3.amazonaws.com/repo.commonsware.com" }
        maven { url 'https://maven.google.com' }
    }
}
```

(from Trips/CW/StaticSecure/build.gradle)

- Add a dependency on `com.commonsware.cwac:netsecurity:0.4.4` to the module(s) that will use CWAC-NetSecurity:

```
dependencies {
  compile 'io.reactivex.rxjava2:rxjava:2.0.2'
  compile 'io.reactivex.rxjava2:rxandroid:2.0.0'
  compile 'com.android.support:recyclerview-v7:26.0.1'
  compile 'com.apollographql.apollo:apollo-rx2-support:0.4.0'
  compile 'com.squareup.okhttp3:okhttp:3.8.1'
  compile 'com.commonsware.cwac:netsecurity:0.4.4'
}
```

(from Trips/CW/StaticSecure/app/build.gradle)

- Add a `<meta-data>` element to teach CWAC-NetSecurity about that same network security configuration XML resource that you are using for Android 7.0+:

```
<application
  android:icon="@drawable/ic_launcher"
  android:label="@string/app_name"
  android:networkSecurityConfig="@xml/network_security"
  android:theme="@style/Theme.Apptheme">
  <meta-data
```

233

```
    android:name="android.security.net.config"
    android:resource="@xml/network_security" />
  <activity
    android:name=".MainActivity"
    android:label="@string/app_name">
    <intent-filter>
      <action android:name="android.intent.action.MAIN" />

      <category android:name="android.intent.category.LAUNCHER" />
    </intent-filter>
  </activity>
</application>
```

(from Trips/CW/StaticSecure/app/src/main/AndroidManifest.xml)

- Create a `TrustManagerBuilder` and call `withManifestConfig()` on it to have it use the network security configuration XML resource from that `<meta-data>` element
- Use `OkHttp3Integrator` to apply CWAC-NetSecurity's `TrustManagerBuilder` setup to an `OkHttpClient.Builder`, then use the `OkHttpClient.Builder` as normal:

```
TrustManagerBuilder tmb=
  new TrustManagerBuilder().withManifestConfig(getActivity());
OkHttpClient.Builder okb=new OkHttpClient.Builder();

try {
  OkHttp3Integrator.applyTo(tmb, okb);
  ApolloClient apolloClient=ApolloClient.builder()
    .okHttpClient(okb.build())
    .serverUrl("https://graphql-demo.commonsware.com/0.3/graphql")
    .build();

  observable=Rx2Apollo.from(apolloClient.query(new GetAllTrips()).watcher())
    .subscribeOn(Schedulers.io())
    .map(response -> (getAllTripsFields(response)))
    .cache()
    .observeOn(AndroidSchedulers.mainThread());
}
catch (Exception e) {
  Toast.makeText(getActivity(), "Um, we crashed!", Toast.LENGTH_LONG).show();
  Log.e(getClass().getSimpleName(), "Exception initializing network", e);
}
```

(from Trips/CW/StaticSecure/app/src/main/java/com/commonsware/graphql/trips/simple/SimpleTripsFragment.java)

NetCipher

In some cases, your users may be at risk merely by connecting to your server. HTTPS hides the data contained in your app's communications with the server, but it does

234

not hide the actual connection with the server itself.

In other cases, your users may have difficulty connecting with your server due to national firewall blocks (e.g., when countries elect to ban certain server types, such as social networks).

To help with these cases, the Guardian Project maintains [NetCipher](#), and [Orbot](#), to help Android apps use the Tor "onion routing" network. NetCipher has specific hooks to help you configure OkHttp to use an Orbot-supplied proxy server to be able to route your HTTPS communications over Tor.

Testing GraphQL

When we make requests of a GraphQL-powered Web service, it would be nice if our app worked.

Part of ensuring that involves writing tests capable of being run repeatedly, particularly after we make code changes, to confirm that everything works now as it did before those changes. There are a variety of ways of going about writing such tests, and in this chapter, we will explore a few of them, in the context of the `Trips/CW/StaticTest` sample project.

This chapter is focused on GraphQL-specific issues with testing, particularly when using Apollo-Android. See *The Busy Coder's Guide to Android Development* for much more extensive coverage of testing.

Testing Using a Real Server

In many cases, the simplest thing to do is to test your code against a real server. This might be a dedicated test or staging GraphQL server, rather than your production server.

If you are using OkHttp directly, you could perform synchronous GraphQL requests, using `execute()` instead of `enqueue()`, as instrumentation test methods are invoked on background threads. Apollo-Android, though, wants to work asynchronously, and that makes things a bit tricky for testing, as you need to block the test method waiting for the work to complete.

There are a variety of ways of implementing this. Since we are using RxJava 2, a simple approach is to use `blockingFirst()`, which will convert a normally-asynchronous RxJava observer chain into a blocking call:

237

```
@Test
public void realServer() throws InterruptedException {
  ApolloClient apolloClient=ApolloClient.builder()
    .okHttpClient(new OkHttpClient())
    .serverUrl("https://graphql-demo.commonsware.com/0.3/graphql")
    .build();

  assertResponse(Rx2Apollo.from(apolloClient.query(new GetAllTrips()).watcher())
    .subscribeOn(Schedulers.io())
    .blockingFirst());
}
```

The `Rx2Apollo` setup, through `subscribeOn()`, is the same as what we might use in the production code. However, rather than setting up an observer and arranging to observe on the main application thread, we just call `blockingFirst()`. This will block until the background thread starts generating results and return whatever the first result is. This gives us an Apollo-Android `Response` object for our `GetAllTrips.Data`, which we can then test to whatever degree we wish. In this sample, the tests are fairly minimal, just enough to prove that this works:

```
private void assertResponse(Response<GetAllTrips.Data> response) {
  assertData(response.data());
  assertEquals(0, response.errors().size());
}

private void assertData(GetAllTrips.Data data) {
  assertEquals(2, data.allTrips().size());
}
```

In this test, we happen to test against the "production" server, to whatever extent this demo server is really "production". Normally, you would test against some other server, supplying its URL to the `ApolloClient`.

Mocking the Apollo-Android Response

Testing against a real server works but has limitations:

- Unless each developer or tester has their own test server, you can run into conflicts when two people test simultaneously, as one might modify the data that is being used by the other
- It requires real network overhead, slowing down test execution
- Testing some scenarios may be difficult or impossible, because you cannot

get the server to respond in the way that you want, due to limitations in the test server itself

Another approach for testing is to use a mocking library to set up mock responses from Apollo-Android, such as by using [Mockito](#).

```java
@Test
public void mockResponse() {
  GetAllTrips.AllTrip firstTrip=Mockito.mock(GetAllTrips.AllTrip.class);
  GetAllTrips.AllTrip secondTrip=Mockito.mock(GetAllTrips.AllTrip.class);
  ArrayList<GetAllTrips.AllTrip> allTrips=new ArrayList<>();

  allTrips.add(firstTrip);
  allTrips.add(secondTrip);

  GetAllTrips.Data data=Mockito.mock(GetAllTrips.Data.class);

  doAnswer(new Answer() {
    @Override
    public Object answer(InvocationOnMock invocation) throws Throwable {
      return(allTrips);
    }
  }).when(data).allTrips();

  assertData(data);
}
```

(from [Trips/CW/StaticTest/app/src/androidTest/java/com/commonsware/graphql/trips/simple/statictest/test/TripTests.java](#))

Here, we:

- Create a pair of do-nothing `GetAllTrip.AllTrip` mock objects
- Pour those into an `ArrayList`
- Create a mock `GetAllTrips.Data` mock object
- Teach that mock to return the `ArrayList` in response to the `allTrips()` method
- Test the result

In this case, the test is fairly pointless, as all we are doing is testing to see if our mocks do what they were told to do. In a larger app, we might pass the mock `GetAllTrips.Data` object to some business logic, then test the results of that logic.

Note that we are mocking the `GetAllTrips.Data`, but not the `Response` object. Apollo-Android marks the `Response` class as `final`, and Mockito cannot mock `final`

239

classes.

Mocking the OkHttp Response

As was discussed earlier in the book, OkHttp has a powerful interceptor framework, which can be used to monitor or even modify HTTP requests and responses. One use case for this is in testing: you could plug an interceptor into OkHttp that would return mock responses instead of actually performing HTTP requests. For example, this library offers one such mocking interceptor.

In the case of that specific library, using it for GraphQL will be difficult. It is designed for testing REST-style Web services, where the URL can be used to distinguish one mock response from another. With GraphQL, there is only one URL.

However, it is possible that future libraries of this type will become available that might be easier to use for GraphQL testing.

Mocking the Server Response

Yet another possibility is to have a mock Web server, one that responds to HTTP, but is embedded in your app and is configurable to return specific responses to specific requests.

The leader in this space is WireMock. Sam Edwards has instructions on using it in Android. However, WireMock is designed for use in testing Java on desktops and servers, and it is rather large for an Android app. In particular, it is likely to cause your app, when tested, to blow past the "64K DEX method reference limit" and force you to implement "multidex" solutions to have such a large app.

An offshoot of the OkHttp project is its `MockWebServer`, which has far fewer features than does WireMock, but it is also a much smaller library.

To use `MockWebServer`, you need to add its dependency to your `androidTest` configuration:

```
androidTestCompile 'com.squareup.okhttp3:mockwebserver:3.8.1'
```

(from Trips/CW/StaticTest/app/build.gradle)

Then, you can start using `MockWebServer` in test methods:

```
@Test
public void mockServer() throws IOException, InterruptedException {
  MockWebServer server=new MockWebServer();

  server.enqueue(new MockResponse().setBody(MOCK_RESPONSE));
  server.start();

  try {
    HttpUrl url=server.url("/0.3/graphql");
    ApolloClient apolloClient=ApolloClient.builder()
      .okHttpClient(new OkHttpClient())
      .serverUrl(url.toString())
      .build();

    assertResponse(Rx2Apollo.from(apolloClient.query(new GetAllTrips()).watcher())
      .subscribeOn(Schedulers.io())
      .blockingFirst());
  }
  finally {
    server.shutdown();
  }
}
```

(from [Trips/CW/StaticTest/app/src/androidTest/java/com/commonsware/graphql/trips/simple/statictest/test/TripTests.java](#))

Here, we:

- Create a `MockWebServer`
- Create a `MockResponse`, indicating that we want the mock server to return the contents of a `MOCK_RESPONSE String` on the next HTTP request made of this server
- Start the server
- Obtain a URL for the server, taking into account whatever port number it wound up choosing (based on what ports are unused)
- Teach Apollo-Android to access that server, instead of some more "real" Web server
- Make the same sort of RxJava-based request as we did when testing against a real Web service
- Shut down the `MockWebServer` when we are done with it

The tricky part is in the `MOCK_RESPONSE`. This is just the HTTP response body, which in our case is plain JSON. You might think that you could just paste in the output from GraphiQL or the equivalent, based on whatever GraphQL document you are using. And, if you are using OkHttp directly, that approach will work.

However, Apollo-Android adds `__typename` fields to each field set that your GraphQL document requests, and your JSON needs to reflect that. So, while our document is:

```
query getAllTrips {
  allTrips {
    id
    title
    startTime
    priority
    duration
    creationTime
  }
}
```

what Apollo-Android actually sends to the server is more like:

```
query getAllTrips {
  allTrips {
    __typename
    id
    title
    startTime
    priority
    duration
    creationTime
  }
}
```

And so our response needs to include __typename values:

```
private static final String MOCK_RESPONSE="{\n"+
"  \"data\": {\n"+
"    \"allTrips\": [\n"+
"      {\n"+
"        \"id\": \"2c494055-78bc-430c-9ab7-19817f3fc060\",\n"+
"        \"title\": \"Vacation!\",\n"+
"        \"startTime\": \"2017-12-20T13:14:00-05:00\",\n"+
"        \"priority\": \"MEDIUM\",\n"+
"        \"duration\": 10080,\n"+
"        \"creationTime\": \"2017-06-15T23:59:43.190Z\",\n"+
"        \"__typename\": \"Trip\"\n"+
"      },\n"+
"      {\n"+
"        \"id\": \"e323fed5-6805-4bcf-8cb6-8b7a5014a9d9\",\n"+
"        \"title\": \"Business Trip\",\n"+
"        \"startTime\": \"2018-01-14T11:45:00-05:00\",\n"+
"        \"priority\": \"HIGH\",\n"+
```

```
"          \"duration\": 4320,\n"+
"          \"creationTime\": \"2017-06-15T23:59:43.190Z\",\n"+
"          \"__typename\": \"Trip\"\n"+
"        }\n"+
"      ]\n"+
"   }\n"+
"}";
```

(from Trips/CW/StaticTest/app/src/androidTest/java/com/commonsware/graphql/trips/simple/statictest/test/TripTests.java)

Otherwise, you will crash inside of Apollo-Android, as it attempts to find the __typename and fails.

Advanced Apollo-Android

Apollo-Android continues to expand its capabilities, to handle broader and deeper GraphQL scenarios in Android apps.

This chapter outlines some additional Apollo-Android features, found starting in v0.4.0 of the library.

Support for Scalar Types

With GraphQL, the server can define [custom scalar types](#). These are marshaled into string values in the JSON. However, if you wish, you can unmarshal them into custom types in your Android app. For example, you could convert GitHub's `DateTime` or `GitTimestamp` scalars into Java `DateTime` objects.

If you are processing the JSON directly (e.g., dynamic GraphQL with OkHttp and Gson), you would use features of your JSON parser for performing this sort of type conversion. However, if you are using Apollo-Android, you can teach the library how to perform conversions on a per-scalar-type basis. Then, anywhere in the generated code that the scalar type appears, your desired data type will be used for the POJOs.

Of course, this requires a little bit of work, as Apollo-Android cannot magically determine how to convert a string into arbitrary sorts of data. Nor does Apollo-Android know which scalars you want to support this way.

The [GitHub/DateTime](#) sample project shows how this is accomplished. This sample app is a clone of the `GitHub/Starred` sample app shown [elsewhere in the book](#). However, it adds the `createdAt` field to the GraphQL query:

```
query myStars($first: Int, $after: String) {
  viewer {
```

245

```
    login
    starredRepositories(first: $first, after: $after, orderBy: {field: STARRED_AT, direction: DESC}) {
      edges {
        cursor
        node {
          id
          name
          createdAt
        }
      }
      pageInfo {
        hasNextPage
      }
    }
  }
}
```

(from GitHub/DateTime/app/src/main/graphql/com/commonsware/graphql/github/api/GitHubApi.graphql)

We want to show that value in the RecyclerView rows.

Gradle Configuration

First, you need to teach the Apollo-Android code generator about the custom scalars that you want to convert and what you want to convert them into. This is done by an apollo closure in your module's build.gradle file, separate from (and outside of) the android closure:

```
apply plugin: 'com.android.application'
apply plugin: 'com.apollographql.android'

dependencies {
  compile 'io.reactivex.rxjava2:rxjava:2.0.2'
  compile 'io.reactivex.rxjava2:rxandroid:2.0.0'
  compile 'com.android.support:recyclerview-v7:26.0.1'
  compile 'com.apollographql.apollo:apollo-rx2-support:0.4.0'
}

android {
    compileSdkVersion 26
    buildToolsVersion "26.0.1"

    defaultConfig {
        minSdkVersion 15
        targetSdkVersion 26
        applicationId "com.commonsware.graphql.github.datetime"
        buildConfigField "String", "GITHUB_TOKEN", '"'+GITHUB_TOKEN+'"'

        jackOptions {
            enabled true
```

246

```
      }
   }

  compileOptions {
    sourceCompatibility JavaVersion.VERSION_1_8
    targetCompatibility JavaVersion.VERSION_1_8
  }
}

apollo {
  customTypeMapping['DateTime']="java.util.Date"
  useSemanticNaming = false
}
```

(from GitHub/DateTime/app/build.gradle)

customTypeMapping is a Map-style data structure, listing the scalar type conversions that you want to apply. The key (DateTime) is the name of the scalar type defined by the server. In our case, we want to convert GitHub's DateTime scalar type. The value (java.util.Date) is the fully-qualified class name of the Java class to use for that type, instead of String.

Implementing Adapters

While the customTypeMapping configuration tells Apollo-Android that you want to convert the scalar to some specific Java type, it does not tell Apollo-Android *how* you want to perform that conversion. That is handled by a CustomTypeAdapter implementation, akin to similar ones in other libraries (Room, Gson, etc.).

A CustomTypeAdapter needs to implement two methods: encode() and decode(). encode() converts from your requested type into a String to use for creating the JSON to send to the GraphQL endpoint. decode() takes the String and converts it into your requested data type. The String representations should match what the GraphQL endpoint uses.

This project implements an ISO8601Adapter that converts dates encoded in the ISO 8601 format into Java Date objects, and vice versa:

```
package com.commonsware.graphql.github;

import com.apollographql.apollo.CustomTypeAdapter;
import java.text.ParseException;
import java.text.SimpleDateFormat;
import java.util.Date;

class ISO8601Adapter implements CustomTypeAdapter<Date> {
```

247

```java
  private static final SimpleDateFormat ISO8601=
    new SimpleDateFormat("yyyy-MM-dd'T'HH:mm:ssZ");

  // based in part on https://stackoverflow.com/a/10621553/115145

  @Override
  public Date decode(String value) {
    try {
      return(ISO8601.parse(value.replace("Z", "+00:00")));
    }
    catch (ParseException e) {
      throw new IllegalArgumentException(value+" is not a valid ISO 8601 date", e);
    }
  }

  @Override
  public String encode(Date value) {
    return(ISO8601.format(value));
  }
}
```

(from [GitHub/DateTime/app/src/main/java/com/commonsware/graphql/github/ISO8601Adapter.java](#))

This does not handle the complete ISO 8601 specification, such as microseconds, due to the fact that Java's date/time handling is a mess. But, this implementation suffices for use with GitHub's GraphQL API.

Registering Adapters

Then, as part of setting up the ApolloClient, we need to call addCustomTypeAdapter() on the ApolloClient.Builder:

```java
  private ApolloClient apolloClient=ApolloClient.builder()
    .okHttpClient(ok)
    .addCustomTypeAdapter(CustomType.DATETIME, new ISO8601Adapter())
    .serverUrl("https://api.github.com/graphql")
    .build();
```

(from [GitHub/DateTime/app/src/main/java/com/commonsware/graphql/github/ViewerFragment.java](#))

addCustomTypeAdapter() takes two parameters:

- A code-generated identifier on the CustomType class, based upon the scalar type name, converted into all-caps (so GitHub's DateTime turns into CustomType.DATETIME)
- An instance of your CustomTypeAdapter for this type

Then, you can use createdAt in the more-natural Date type rather than as a simple String:

```java
private static class RowHolder extends RecyclerView.ViewHolder {
  final private TextView name;
  private final DateFormat dateFormat;

  RowHolder(View itemView, DateFormat dateFormat) {
    super(itemView);

    name=(TextView)itemView.findViewById(android.R.id.text1);
    this.dateFormat=dateFormat;
  }

  void bind(MyStars.Node node) {
    name.setText(String.format("%s (%s)", node.name(),
      dateFormat.format(node.createdAt())));
  }
}
```

(from GitHub/DateTime/app/src/main/java/com/commonsware/graphql/github/ViewerFragment.java)