



C o m m u n i t y E x p e r i e n c e D i s t i l l e d

Learning GraphQL and Relay

Build data-driven React applications with ease using GraphQL and Relay

Samer Buna

www.allitebooks.com

[PACKT] open source*

community experience distilled

Learning GraphQL and Relay

**Build data-driven React applications with ease using
GraphQL and Relay**

Samer Buna



BIRMINGHAM - MUMBAI

Learning GraphQL and Relay

Copyright © 2016 Packt Publishing

All rights reserved. No part of this book may be reproduced, stored in a retrieval system, or transmitted in any form or by any means, without the prior written permission of the publisher, except in the case of brief quotations embedded in critical articles or reviews.

Every effort has been made in the preparation of this book to ensure the accuracy of the information presented. However, the information contained in this book is sold without warranty, either express or implied. Neither the author, nor Packt Publishing, and its dealers and distributors will be held liable for any damages caused or alleged to be caused directly or indirectly by this book.

Packt Publishing has endeavored to provide trademark information about all of the companies and products mentioned in this book by the appropriate use of capitals. However, Packt Publishing cannot guarantee the accuracy of this information.

First published: August 2016

Production reference: 1250816

Published by Packt Publishing Ltd.

Livery Place

35 Livery Street

Birmingham

B3 2PB, UK.

ISBN 978-1-78646-575-7

www.packtpub.com

Credits

Author

Samer Buna

Copy Editor

Safis Editing

Reviewers

Hafiz Ismail

Lee Byron

Project Coordinator

Sheejal Shah

Commissioning Editor

Wilson D'souza

Proofreader

Safis Editing

Acquisition Editor

Smeet Thakkar

Indexer

Rekha Nair

Content Development Editor

Divij Kotian

Production Coordinator

Melwyn Dsa

Technical Editor

Rupali R. Shrawane

About the Author

Samer Buna is a technical content author, software engineer, and mentor. He has a master's degree in information security and over ten years of progressive experience and success creating tailored solutions for businesses within many industries.

Samer is passionate about everything JavaScript, and he loves exploring new libraries. His favorite technical stacks are Node.js for the backend and React.js for the frontend.

Samer has authored a few books and online courses about React and GraphQL. You can follow him on Twitter at [@samerbuna](https://twitter.com/samerbuna), and you can read more of what he writes at <https://edgecoders.com/>.

Acknowledgments

First and foremost, I would like to thank my wife, Chalena, for standing beside me throughout my career and the writing of this book. Not only does she take care of the world around me and allow me to focus 100% on what I do, but she also often helps me do what I do and review what I write. I dedicate this book to her and our kids, Leo and Ally, who are just about the best children a dad could hope for. Their happy, loving, and fun presence is what keeps me going. I'd like to thank my friend and genius reviewer Julia Hunt for all of her excellent contributions to this book. Julia's feedback was great, as always, and it made this book a much better product. I'd like to also thank the Packt Publishing team I was fortunate to work with for their patience and guidance throughout the process of drafting and reviewing this book.

About the Reviewers

Hafiz Ismail is a software engineer, an open source buccaneer, and not an astrophysicist. He writes articles on the latest in web technology at <https://wehavefaces.net>

Lee Byron has been making things at Facebook since 2008: React, GraphQL, Immutable.js, Mobile, JavaScript.

www.PacktPub.com

For support files and downloads related to your book, please visit www.PacktPub.com.

Did you know that Packt offers eBook versions of every book published, with PDF and ePUB files available? You can upgrade to the eBook version at www.PacktPub.com and as a print book customer, you are entitled to a discount on the eBook copy. Get in touch with us at service@packtpub.com for more details.

At www.PacktPub.com, you can also read a collection of free technical articles, sign up for a range of free newsletters and receive exclusive discounts and offers on Packt books and eBooks.



<https://www2.packtpub.com/books/subscription/packtlib>

Do you need instant solutions to your IT questions? PacktLib is Packt's online digital book library. Here, you can search, access, and read Packt's entire library of books.

Why subscribe?

- Fully searchable across every book published by Packt
- Copy and paste, print, and bookmark content
- On demand and accessible via a web browser

Free access for Packt account holders

If you have an account with Packt at www.PacktPub.com, you can use this to access PacktLib today and view 9 entirely free books. Simply use your login credentials for immediate access.

Table of Contents

Preface	1
<hr/>	
Chapter 1: An Introduction to GraphQL and Relay	6
What is GraphQL?	6
What is Relay?	8
Why GraphQL?	10
RESTful APIs versus GraphQL APIs	11
Why Relay?	19
Understanding Relay's core principles	19
Storage and caching	20
Object identification	20
The connection model	21
Setting up a simple GraphQL server	22
Installing Node.js	22
Defining the schema	24
Using the schema	28
Rolling the dice	29
Using field arguments	31
Setting up MongoDB	32
Setting up an HTTP interface	37
The GraphiQL editor	39
Summary	41
<hr/>	
Chapter 2: The Query Language	42
Documents and operations	42
Fields	44
Variables	45
Directives	47
Aliases	49
Fragments	51
Mutations	53
Summary	54
<hr/>	
Chapter 3: The GraphQL Schema	55
The schema object	55
Introspection	59
The type system	65

Scalars and object types	66
Interfaces and unions	66
Type modifiers	70
Enums	72
The resolve function	73
First argument – source	74
Second argument – args	74
Third argument – context	78
Fourth argument – info	79
Resolving with promises	80
Validation	85
Versioning	86
Summary	88
Chapter 4: Configuring React Applications to Use Relay	89
The example GraphQL schema	89
The quotes library	92
Setting up Webpack	94
Using GraphQL without Relay in React applications	98
Relay containers	101
Summary	104
Chapter 5: Making GraphQL Queries Relay-Compliant	105
Transforming GraphQL queries for Relay	105
Root-level field for the quotes library	113
First query operation with Relay	117
Relay's connection model	122
Summary	131
Chapter 6: Relay Variables and Object Identification	132
Implementing search	132
Adding a search feature to the GraphQL API	133
Implementing the search feature in Relay	135
Adding a search form component	135
Using Relay variables	136
Implementing likes	140
Relay's global ID	146
The Node interface	148
Summary	152
Chapter 7: Relay Mutations	154
Relay-compliant GraphQL mutations	154

Relay.Mutation	160
Optimistic updates	166
Summary	169
Chapter 8: Deploying to the Cloud	170
Preparing for deployment	170
Creating a GitHub repository	170
Deploying locally	173
Deploying on EC2	178
Installing MongoDB	184
Running the Node.js server	185
Deploying on Heroku	188
Cloud-hosted MongoDB	192
Summary	198
Index	199

Preface

There's a new choice for implementing APIs—the open source and Facebook-created GraphQL specification. Designed to solve many of the issues of working with REST, GraphQL comes alongside RelayJS, a React library for querying a server that implements the GraphQL specification. This book takes you quickly and simply through the skills you need to be able to build production-ready applications with both GraphQL and RelayJS.

What this book covers

Chapter 1, *An Introduction to GraphQL and Relay*, introduces you to Relay and GraphQL with the assumption that you have never heard of either. We will also discuss the problems that GraphQL and Relay aim to solve. Lastly, you will learn how to configure your Node.js application for GraphQL by setting up a Node.js GraphQL endpoint with a basic Hello World example.

Chapter 2, *The Query Language*, teaches you the syntax of the GraphQL language and the different features that are available there.

Chapter 3, *The GraphQL Schema*, gives you hands-on experience for working a GraphQL schema and the core features of a GraphQL runtime.

Chapter 4, *Configuring React Applications to Use Relay*, walks you through the steps needed to get your React application ready for use with Relay.

Chapter 5, *Making GraphQL Queries Relay-Compliant*, explains why just having a GraphQL server is not enough. The server will need to be Relay-compliant. This chapter will walk you through the necessary steps to make it so.

Chapter 6, *Relay Variables and Object Identification*, is about how a user can query the GraphQL endpoint with Relay at the React application's entry point, reducing round trips to the GraphQL server.

Chapter 7, *Relay Mutations*, will teach you how to invoke changes by invoking mutations with Relay on the GraphQL endpoints.

Chapter 8, *Deploying to the Cloud*, walks you through deploying your GraphQL/Relay app on EC2 and Heroku.

What you need for this book

The following are the requirements:

- Node 6.x
- Git
- A cup of coffee

Who this book is for

This book does not assume that you have any prior experience or familiarity with GraphQL or Relay. You should, however, be comfortable writing Node.js applications on a MongoDB database with REST APIs as well as applications on the client side using React and ES2015.

Conventions

In this book, you will find a number of text styles that distinguish between different kinds of information. Here are some examples of these styles and an explanation of their meaning.

Code words in text, database table names, folder names, filenames, file extensions, pathnames, dummy URLs, user input, and Twitter handles are shown as follows: "We can include other contexts through the use of the `include` directive."

A block of code is set as follows:

```
const queryType = new GraphQLObjectType({  
  name: 'RootQuery',  
  fields: {  
    hello: {  
      type: GraphQLString,  
      resolve: () => 'world'  
    }  
  }  
});
```

When we wish to draw your attention to a particular part of a code block, the relevant lines or items are set in bold:

```
import socket  
socket.settimeout(3)  
newSocket = socket.socket()  
newSocket.connect(("localhost", 22))
```

Any command-line input or output is written as follows:

```
~ $ curl -o-
https://raw.githubusercontent.com/creationix/nvm/v0.31.4/install.sh | bash
```

New **terms** and **important words** are shown in bold. Words that you see on the screen, for example, in menus or dialog boxes, appear in the text like this: "Click on the **OS X** link."

Warnings or important notes appear in a box like this.



Tips and tricks appear like this.



Reader feedback

Feedback from our readers is always welcome. Let us know what you think about this book—what you liked or disliked. Reader feedback is important for us as it helps us develop titles that you will really get the most out of.

To send us general feedback, simply e-mail feedback@packtpub.com, and mention the book's title in the subject of your message.

If there is a topic that you have expertise in and you are interested in either writing or contributing to a book, see our author guide at www.packtpub.com/authors.

Customer support

Now that you are the proud owner of a Packt book, we have a number of things to help you to get the most from your purchase.

Downloading the example code

You can download the example code files for this book from your account at <http://www.packtpub.com>. If you purchased this book elsewhere, you can visit <http://www.packtpub.com/support> and register to have the files e-mailed directly to you.

You can download the code files by following these steps:

1. Log in or register to our website using your e-mail address and password.
2. Hover the mouse pointer on the **SUPPORT** tab at the top.
3. Click on **Code Downloads & Errata**.
4. Enter the name of the book in the **Search** box.
5. Select the book for which you're looking to download the code files.
6. Choose from the drop-down menu where you purchased this book from.
7. Click on **Code Download**.

Once the file is downloaded, please make sure that you unzip or extract the folder using the latest version of:

- WinRAR / 7-Zip for Windows
- Zipeg / iZip / UnRarX for Mac
- 7-Zip / PeaZip for Linux

The code bundle for the book is also hosted on GitHub at

<https://github.com/PacktPublishing/Learning-GraphQL-and-Relay>. We also have other code bundles from our rich catalog of books and videos available at <https://github.com/PacktPublishing/>. Check them out!

Downloading the color images of this book

We also provide you with a PDF file that has color images of the screenshots/diagrams used in this book. The color images will help you better understand the changes in the output.

You can download this file from

http://www.packtpub.com/sites/default/files/downloads/LearningGraphQLAndRelay_ColorImages.pdf.

Errata

Although we have taken every care to ensure the accuracy of our content, mistakes do happen. If you find a mistake in one of our books—maybe a mistake in the text or the code—we would be grateful if you could report this to us. By doing so, you can save other readers from frustration and help us improve subsequent versions of this book. If you find any errata, please report them by visiting <http://www.packtpub.com/submit-errata>, selecting your book, clicking on the **Errata Submission Form** link, and entering the details of your errata. Once your errata are verified, your submission will be accepted and the errata will be uploaded to our website or added to any list of existing errata under the Errata section of that title.

To view the previously submitted errata, go to <https://www.packtpub.com/books/content/support> and enter the name of the book in the search field. The required information will appear under the **Errata** section.

Piracy

Piracy of copyrighted material on the Internet is an ongoing problem across all media. At Packt, we take the protection of our copyright and licenses very seriously. If you come across any illegal copies of our works in any form on the Internet, please provide us with the location address or website name immediately so that we can pursue a remedy.

Please contact us at copyright@packtpub.com with a link to the suspected pirated material.

We appreciate your help in protecting our authors and our ability to bring you valuable content.

Questions

If you have a problem with any aspect of this book, you can contact us at questions@packtpub.com, and we will do our best to address the problem.

1

An Introduction to GraphQL and Relay

GraphQL provides a common interface between client and server applications for fetching and manipulating data. **Relay** is a JavaScript framework that uses GraphQL to enable React applications to communicate their data requirements in a declarative way.

This chapter will be an introduction to GraphQL and Relay. Here are the topics that we will cover in this chapter:

- What is GraphQL, what is Relay, and what problems do they solve?
- What is declarative data communication?
- What does GraphQL look like and how does it work on the server?
- What is a GraphQL schema?
- How does GraphQL work with databases such as MongoDB?
- How can GraphQL be used over HTTP?

What is GraphQL?

GraphQL is defined as a data query language and runtime. It's important to understand the two different parts of this definition:

- GraphQL is a **language**. If we teach it to a software client application, that application will be able to declaratively communicate its data requirements to a backend data service that also speaks GraphQL.

- GraphQL is a **runtime**. It's an execution layer a server application can use to understand and respond to any requests made with the GraphQL language. Think of this layer as simply a translator of the GraphQL language, or a GraphQL-speaking agent who represents the data service.

Anyone can invent a new language and start speaking it, but no one would understand them without learning the new language first, or having someone translate it for them. That's why we need to implement a runtime for GraphQL on the backend servers.

Backend servers speak their own languages, and they often speak multiple languages. A typical modern API server speaks at least two languages: one for the database, such as SQL, and another for processing the data, such as Java or Ruby.

GraphQL is designed to play well with other backend languages. We can implement GraphQL as a layer on top of any existing server logic. This layer will enable the server to understand GraphQL requests and pass them down to its existing logic to fetch data and fulfill the original requests.

Regardless of whether we're writing GraphQL requests for client applications, or GraphQL runtimes for server applications, we need to learn and understand the principles and standards of the GraphQL language first.

Learning the GraphQL language itself will be simple. It's not a big language, and it's very close to **JavaScript Object Notation (JSON)**. If you're comfortable with JSON, you'll be able to pick up the GraphQL language in no time. If you don't know JSON, I'd recommend that you do some reading about it first. The *Introducing JSON* article at <http://json.org/> is an excellent start.

Once we are fluent in the GraphQL language itself, we can use this new skill to translate for web and mobile applications. These applications have a constant need to communicate their data requirements to data services, and we can use the GraphQL language to do that for them.



Just like a child can learn a new language fast, while an adult will have a harder time picking it up, starting a new application from scratch using GraphQL will be a lot easier than introducing GraphQL to a mature application.

The GraphQL runtime layer, which can be written in any language, defines a generic graph-based schema to publish the capabilities of the data service it represents. Client applications can query the schema within its capabilities. This approach decouples clients from servers and allows both of them to evolve and scale independently.

A GraphQL operation can be either a query (*read* operation), or a mutation (*write* operation). For both cases, the operation is a simple string that a GraphQL service can parse and respond to with data in a specific format. The popular response format that is usually used for mobile and web applications is JSON.

Here's an example of a GraphQL query that a client can use to ask a server about the name and e-mail of user #42:

```
{  
  user(id: 42) {  
    firstName  
    lastName  
    email  
  }  
}
```

Here's a possible JSON response for that query:

```
{  
  "data": {  
    "user": {  
      "firstName": "John",  
      "lastName": "Doe",  
      "email": "john@example.com"  
    }  
  }  
}
```

The request and response in a GraphQL communication are related. A request determines the shape of its response, and a response can be used to easily construct a suitable request.

GraphQL on the server is just a specification that defines various design principles, including a hierarchical structure that supports composition, support of arbitrary code, a strong type system, introspective nature, and many more.

What is Relay?

Relay is defined as a framework for building data-driven React applications. It's important to understand the two aspects of Relay in this definition:

- Relay is a **framework**, and it's actually an opinionated one. Opinionated frameworks make most of the design decisions for us, and they want us to do things a certain way. Depending on our requirements and conditions, this could be a great thing, but it could also be a great limitation. This applies to all

frameworks, not just Relay.

- Relay is for **data-driven applications**. When writing applications that are not driven by data, or applications that have minimal data interactions, Relay might not be the best choice.

However, most frontend applications are data-driven, even if at first we think otherwise. Simplify the problem domain, and it will most likely point to some data requirements.

Relay is a young and developing framework that is constantly changing. Its API syntax will most likely change in the near future; however, its concepts and design principles will stand the test of time.



Although Relay's concepts and principles can be applied to other view libraries, what Facebook open-sourced in 2015 as Relay is an extension to the React.js library, and it will only work with that library.

Here's an example React component with its Relay data requirements:

```
const UserCard = ({user}) =>
  <div className="user-card">
    Name : {user.firstName} {user.lastName}
    Email : {user.email}
  </div>

UserCard = Relay.createContainer(UserCard, {
  fragments: {
    user: () => Relay.QL`fragment on User {
      firstName
      lastName
      email
    }
  }
});
```

This `UserCard` React component displays a user's name and e-mail, it requires the existence of a `user` object, and it requires that object to have the following properties: `firstName`, `lastName`, and `email`. Using a GraphQL fragment, we expressed this exact data requirement within a Relay container that wraps the `UserCard` React component. We'll explore GraphQL fragments in [Chapter 2, *The Query Language*](#), and Relay containers in [Chapter 4, *Configuring React Applications to Use Relay*](#).



Although this is a book about Relay and GraphQL together, it is very important to understand that they are separate projects, and the dependency between them is only unidirectional. Relay depends on GraphQL but a GraphQL server does not need Relay at all. While we can't run a Relay application without a GraphQL server, we can certainly run a GraphQL server without Relay's extensions or clients.

GraphQL is not a part of any web or mobile frameworks; it operates independently, and it can be used by all frameworks. The first clients that used GraphQL at Facebook were the iOS and Android applications, a long time before Relay was born.

Why GraphQL?

Data communication is probably the most important activity in a software application. When was the last time you developed an application that did not communicate with a data service?

Even what we used to call *static sites* are now starting to be based on generators that use data. Games and other applications that load their data with their initial download also require some form of data communication afterwards to save preferences, track usages, and keep records about everything the user is doing. With the foreseen future of the *Internet of Things*, where micro devices will be everywhere, the role of data communication will become more important.

Relational databases successfully deliver on reliability, consistency, and integrity for the task of storing data. Document databases give us flexibility to manage document-oriented information, and scale it horizontally. However, the current solutions we use for communicating data between multiple software applications have many problems. We came up with all sorts of interfaces between applications and data services to fill the gaps. The most popular interfaces we use today are RESTful APIs and adhoc HTTP APIs. These interfaces are especially popular for web applications, but they've seen success with mobile applications as well.

Here are some of the tasks an **Application Programming Interface (API)** can do:

- Act as a controller between protected raw data services and software clients
- Parse a client request for access rights and enforce them
- Construct SQL join statements to satisfy a client request efficiently
- Process raw data into structures demanded by clients
- Respond with data in specific formats such as JSON or XML

RESTful APIs versus GraphQL APIs

RESTful APIs are widely popular and have excellent use cases, but they also have limitations and disadvantages. They come with some dependencies on browser implementations of HTTP, and different browsers have different support for HTTP methods, and different interpretation for HTTP response codes. Using only HTTP methods and response codes limits what we can do with RESTful APIs and developers usually resort to customizing and interpreting the request payload instead.

In RESTful APIs, the language we use for the request is different than the language we use for the response. There is a disconnect between the request and the response, just like there is a disconnect between a question in English and an answer to that question in Japanese. There are no standards or agreements about what request and response HTTP codes mean and implementers use different specifications, which makes working with different APIs unpredictable. This lack of standards negatively affects the learning and development process around these APIs, and makes consuming them a challenge. Without standard specifications, developers need to consult documentation to understand the approach taken by every provider, and documentation is always at the risk of becoming outdated.

To consume RESTful APIs, we use a URL to read from or write to a single resource, such as a product, a person, a post, or a comment. If we need to work with multiple resources such as a list of posts with a list of comments, we need to use multiple endpoints. Alternatively, we can develop a *custom* endpoint (given that we have access to do so). The clients do not have any control over the response, unless we start customizing those endpoints to support that control. For example, we can't ask a friend resource endpoint for just the name and location of a friend, we can only ask for all the information about that friend, whether we need it or not. The clients basically depend on the servers, and this fact limits their growth because it will be tied to the growth of the servers.

Some of these issues are solved by other application programming interfaces such as JSON APIs, JSend, JSON LD, and many more. GraphQL is one other alternative that is attempting to solve most of these issues.

First, a draft GraphQL RFC specification has been created. It's managed by Facebook, but it's open source on GitHub and anyone can contribute to it. All GraphQL implementers are expected to honor that specification and work with the community to update the specification when needed.

GraphQL is protocol-agnostic and does not depend on anything HTTP. We don't use HTTP methods or HTTP response codes with GraphQL. However, HTTP is one channel where we can do GraphQL communication, and it will naturally be the popular channel for web development.

The language used for a GraphQL request is directly related to the language used for the response. If we analyze a JSON response, we'll find it to be a dictionary that has keys and values. The values can themselves be nested dictionaries with their own keys and values, or with arrays of values. Since the values basically represent the data, if we strip out all the values from the JSON dictionary, we get the GraphQL query that can be used for the request. This is a simple question-to-answer relationship that's expressed naturally with the same language. Since we use a similar language to communicate between clients and servers, debugging problems become easier. Furthermore, the GraphQL specification adopts a strong type system for all GraphQL elements; any misuse can be easily detected and properly reported. Also, with GraphQL queries mirroring the shape of their response, any deviations can be detected, and these deviations would point us to the exact query fields that are not resolving correctly.

A GraphQL server can be a single endpoint that handles all the client requests, and it can give the clients the power to customize those requests at any time. Clients can ask for multiple resources in the same request and they can customize the fields needed from all of them. This way, clients can be in control of the data they fetch and they can easily avoid the problems of over-fetching and under-fetching. With GraphQL, clients and servers are independent and they can be changed without affecting each other.

These are some of the reasons that make GraphQL efficient, effective, and easy to use. However, the most important reason why GraphQL is considered a game changer is its mental model around *declarative data communication*.

The idea of GraphQL was born out of practical needs, and these needs were mainly centered around mobile clients. Here are some examples of these needs:

- Mobile clients are smart and have evolving data requirements, and we can't have them depend on a data service. We need to give them more power and have them decide what data to consume.
- We can't control the versions of mobile applications like we do on the Web. We need a way to add new features without removing the old ones, and we need a way to communicate what features are now deprecated.
- Resources available for mobile clients are limited. We can't entertain the idea of multiple round-trips to the server to collect the data required by a single view, and we need to minimize any processing needed to piece together data returned by servers.

However, I'd argue that the most important need that influenced the creation of GraphQL was not mobile-specific, but rather one that applies to all platforms:

- The developer experience is as important as, and maybe actually more important than, the user experience. DI/DX is becoming the new UI/UX because the former drives the latter. When it comes to data communication, this means we need to abstract the imperative steps needed to communicate an application's data requirements, and give developers a declarative language for that instead. This language should enable developers to express their applications' data requirements in a way close to how that data will actually be used in their applications.

Putting the product developers' needs first means that instead of thinking about the proper ways to expose data on the servers, we first think about the developers who build frontend applications, and the proper ways for them to express their applications' data requirements.

That's why frontend application developers will love GraphQL. From their point of view, it's a query language that allows them to ask for the data required by their applications in a simple, natural, and declarative way that mirrors the way they use that data in their applications.

The needs that influenced GraphQL are really best explained with an example. Let's imagine that we are the developers responsible for building a shiny new user interface to represent the Star Wars films and characters.

The first UI we've been tasked to build is simple: a view to show information about a single Star Wars person, for example, Darth Vader. The view should display the person's name, birth year, planet name, and the titles of all the films in which they appeared.

As simple as that sounds, we're actually dealing with three different resources here: `person`, `planet`, and `film`. The relation between these resources is simple and anyone can guess the shape of the data here. A `person` object belongs to one `planet` object, and it will have one or more `film` objects.

The JSON data for this UI could be something like:

```
{  
  "data": {  
    "person": {  
      "name": "Darth Vader",  
      "birthYear": "41.9BBY",  
      "planet": {  
        "name": "Tatooine"  
      },  
    },  
  },  
}
```

```
  "films": [
    { "title": "A New Hope" },
    { "title": "The Empire Strikes Back" },
    { "title": "Return of the Jedi" },
    { "title": "Revenge of the Sith" }
  ]
}
}
```

Assuming a data service gave us this exact structure for the data, here's one possible way to represent its view with React.js:

```
// The Container Component:
<PersonProfile person={data.person}></PersonProfile>

// The PersonProfile Component:
Name: {person.name}
Birth Year: {person.birthYear}
Planet: {person.planet.name}
Films: {person.films.map(film => film.title)}
```

This is a simple example, and while our experience with Star Wars might have helped us here a bit, the relationship between the UI and the data is clear. The UI used all the *keys* from the assumed JSON data object.

Let's now see how we can ask for this data using a RESTful API.

We need a single person's information, and assuming that we know the `id` of that person, a RESTful API is expected to expose that information with an endpoint like:

```
/people/{id}
```

This request will give us the name, birth year, and other information about the person. A good API will also give us the ID of this person's planet, and an array of IDs for all the films in which this person appeared.



Instead of IDs, RESTful APIs will usually give us the ready URLs that we need to follow to fetch more information about the resource. I am using IDs in the following examples to simplify the concept.

The JSON response for this request could be something like:

```
{  
  "name": "Darth Vader",  
  "birthYear": "41.9BBY",  
  "planetId": 1,  
  "filmIds": [1, 2, 3, 6],  
  *** other information we do not need for this view ***  
}
```

Then to read the planet's name, we ask:

```
/planets/1
```

And to read the film titles, we ask:

```
/films/1  
/films/2  
/films/3  
/films/6
```

Once we have all six responses from the server, we can combine them to satisfy the data needed by our view.

Besides the fact that we had to do six round trips to satisfy a simple data need for a simple UI, our approach here was imperative. We gave instructions for how to fetch the data and how to process it to make it ready for the view.



A RESTful API for Star Wars data is currently hosted at <http://swapi.co/>. Go ahead and try to construct our data person object there; the field names might be a bit different, but the API endpoints should be the same. You will need to do exactly six API calls. Furthermore, you will have to over-fetch information that the view does not need.

Of course, this is just one implementation of a RESTful API for this data. There could be better implementations that will make this view easier to implement. For example, if the API server implemented nested resources and understood the relation between a person and a film, we could read the films data with:

```
/people/{id}/films
```

However, API developers don't usually implement nested resource endpoints by default, and we would need to ask them to create these custom endpoints for us when we need them. That's the reality of scaling this type of API, we just add custom endpoints to efficiently satisfy the growing clients' needs. Managing custom endpoints like these without a structure around them would be a challenge.

Let's now look at the GraphQL approach. GraphQL on the server embraces the custom endpoints idea and takes it to its extreme. The server will be a single endpoint that replies to all data requests, and the interface channel does not matter. If we use an HTTP interface, HTTP methods and response codes would not matter either.

Let's assume we have a single GraphQL endpoint exposed over HTTP at `/graphql`. Since we want to ask for the data we need in a single round-trip, we'll need a way to express our complete data requirements for the server. We do this with a GraphQL query:

```
/graphql?query={...}
```

A GraphQL query is just a string, but it will have to include all the pieces of the data that we need, and this is where the declarative power comes in.

In English, here's how we declare our data requirement: we need a person's name, birth year, their planet's name, and the titles of all their films.

In GraphQL, this translates to:

```
{
  person(ID: ...) {
    name
    birthYear
    planet {
      name
    }
    films {
      title
    }
  }
}
```

Read the English-expressed requirements one more time and compare it to the GraphQL query. It's as close as it can get. Now compare this GraphQL query with the original JSON data that we started with:

```
{
  "data": {
    "person": {
      "name": "Darth Vader",
      "birthYear": "41.9BBY",
      "planet": {
        "name": "Tatooine"
      },
      "films": [
        { "title": "A New Hope" },
        { "title": "The Empire Strikes Back" },
        { "title": "Return of the Jedi" }
      ]
    }
  }
}
```

```
{ "title": "Return of the Jedi" },
{ "title": "Revenge of the Sith" }
]
}
}
```

The GraphQL query has the exact structure of the JSON data, except without all the *values* parts. If we think of this in terms of a question-answer relation, the question is the answer statement without the answer part.

If the answer statement is:

The closest planet to the Sun is Mercury.

A good representation of the question is the same statement without the answer part:

(What is) the closest planet to the Sun?

The same relation applies to a GraphQL query. Take a JSON response, remove all the *answer* parts (which are the values), and you'll get a GraphQL query very suitable to represent a question for that JSON response.

Now compare the GraphQL query with the React UI we defined for the data. Everything in the GraphQL query is used in the UI, and every variable used in the UI appears in the GraphQL query. This is the great mental model of GraphQL. The UI knows the exact data it needs, and extracting that requirement is fairly easy. Coming up with a GraphQL query is simply the task of extracting what's used as variables directly from the UI. Also, if we invert this model, it would still hold the power. If we have a GraphQL query, we know exactly how to use its response in the UI, because the query will be the same *structure* as the response. We don't need to inspect the response to know how to use it, and we don't need any documentation for the API; it's all built in.

A GraphQL API for Star Wars data is hosted at <https://github.com/graphql/swapi-graphql>. Go ahead and try to construct our data person object there. There are a few minor differences that we'll explain later, but here's the official query you can use against this API to read our data requirement for the view (with Darth Vader as an example):

```
{
  person(personID: 4) {
    name
    birthYear
    homeworld {
      name
    }
    filmConnection {
```

```
    films {  
      title  
    }  
  }  
}  
}
```

The screenshot shows the GraphiQL interface with a query editor on the left and a results viewer on the right.

Query (Left):

```
1 1 {  
2 2   person(personID: 4) {  
3 3     name  
4 4     birthYear  
5 5     homeworld {  
6 6       name  
7 7     }  
8 8     filmConnection {  
9 9       films {  
10 10         title  
11 11       }  
12 12     }  
13 13   }  
14 14 }  
15 15 }
```

Response (Right):

```
{  
  "data": {  
    "person": {  
      "name": "Darth Vader",  
      "birthYear": "41.9BBY",  
      "homeworld": {  
        "name": "Tatooine"  
      },  
      "filmConnection": {  
        "films": [  
          {  
            "title": "A New Hope"  
          },  
          {  
            "title": "The Empire Strikes Back"  
          },  
          {  
            "title": "Return of the Jedi"  
          },  
          {  
            "title": "Revenge of the Sith"  
          }  
        ]  
      }  
    }  
  }  
}
```

This request gives us a response structure very close to what our views used, and remember, we are getting all of this data in a single server round-trip. We'll explore the **GraphiQL** editor that you see in this API later in the chapter.

Why Relay?

React enabled us to create declarative UI views, and model the state for those views and not the transactions to render them. With React, we simply define views as functions of data.

Working with data, however, was the missing piece in the puzzle. Relay is one option to complete that puzzle and act as the data manager for React applications.

Just like React made declarative programming easier for building user interfaces, Relay can make declarative programming easier for fetching and mutating the data required for these user interfaces.

With Relay, we just declare what we need to happen to the data, and Relay will do the actual steps needed to satisfy our needs.

Working with data is always a challenge. We need to be aware of the performance issues around data communication. Are we making optimal requests, or are we making multiple requests that can be batched and further optimized? What should we do when a request fails? How do we handle errors? Should we retry a failed request? If we retry, when should we give up?

Furthermore, when users interact with a user interface and make some changes, we want them to see their changes reflected right away while we attempt to persist these changes to the database. We want to be able to either confirm the changes or roll them back once we have a response from the server.

There is also the task of paginating data in a smart way that can handle changed items. Think about the case where we fetched 10 items from a list, and before we fetch the next 10 items, an item was added or removed from the 10 items we originally fetched. Both the client and the server need to be aware of these edge cases.

There is also caching on both clients and servers. What can safely go to the cache and what cannot? How do we handle merging of cached data with new data? When do we expire things out of the cache? How do we cache related objects without duplicating them? Think about caching a comment, do we cache it under a post object, or do we cache it under an author object?

Working with data is the type of work that could be handled by a framework, and Relay is an attempt to innovate in that domain.

Understanding Relay's core principles

Here are some of the core principles and design decisions behind Relay.

Storage and caching

Relay uses a single normalized client-side data store in memory called **Relay Store**. When Relay loads up in the browser for the first time, it stores all data in this store, and it manages a simple state for every record in there.

In front of the Relay Store, Relay has a **Queue Store** where it manages the inflight changes to the data. The Queue Store allows us to do things, such as optimistic updates in the user interface. It also allows for easier rollbacks in case a change action fails. Instead of manually managing a state for the action, we can have Relay just remove the faulty object from the Queue Store. Behind the Relay Store, Relay has a Cache layer, which can be any storage engine, such as `localStorage` for example.

The hierarchy of these three layers is important in Relay because the first layer that can resolve a record will resolve that record.

Object identification

All objects in Relay have unique IDs over the entire system. This allows Relay to re-fetch any record when it needs to, and it prevents any ambiguity between objects inside of the Relay Store. Without proper unique IDs, duplicate records will find their way into the data store.

For example, imagine we have a client application where we show a list of users whose name begin with J, and for every user, we show their list of friends. User Jane who happens to also be a friend of John will be loaded twice in that view.

Relay also has a *diffing* algorithm to make data fetching as efficient as possible. If we already have part of an object's data, but need more, we don't need to ask for the whole object, we can ask for only the difference between what we need and what we have.

For example, in the same client application where we show a list of users, the initial list page shows the name and location of users, and when we click on a user record, we want to show their name, location, e-mail, and phone number. When we click on a user who is globally identified with a unique ID, we know that we already have their name and location in memory, so we can ask the data service for just their e-mail and phone number.

If we have another interface where we show a profile picture of the user along with their name and phone number, we would then need to ask the data service only about their profile picture.

The connection model

When we need to paginate a list, we have a few models we can use:

- **The offset/limit model:** Let's say we have a list of comments for a blog post, sorted by creation date, most recent first. We have A through Z, and A is the most recent comment. To fetch the first three comments from that list, we do **offset 0, limit 3**, and we get A, B, and C. Before we ask for the next page, someone added a new comment, which now becomes the most recent comment on top of the list, before A. Our next page will be **offset 3, limit 3** and we'd get C, D, and E. *We have a duplicate C.* There is also the case where after we received A, B, and C, someone deleted their embarrassing comment A, so now the next **offset 3, limit 3** operation will get us E, F, and G, and we would miss D completely. This is not ideal.
- **The after/first model:** For the same list of comments example, the first page, we do **after null, first 3**. We get A, B, and C. If someone deletes A, or adds a new comment before A, our next page would not be affected, because our next page would be **after C, first 3**, so we'll get back D, E, and F. This solves the problem mentioned in the offset/limit model. We need unique IDs for every record to make this model work, because we need to reference the records we see in the *after* value. It's no longer just numbers for pagination. One problem with this model is that there is no way to figure out whether there is more information to fetch or we just fetched the last available slice of comments. This means we need to do an extra request that returns an empty slice to conclude that we were on the last page.
- **The connection model:** Relay extends the after/first model with a structure of *edges and nodes*, so that we can store extra information and metadata about the paginated data. The actual data is represented as nodes within edges. For example, we can use the top-level field to ask for the total number of records, or whether we have a next page or not. Every node gets a built-in *cursor* to identify it, and we can use that cursor in our after/first calls. We don't need to manually manage a unique ID for every record.

Setting up a simple GraphQL server

Since GraphQL is a specification for a server runtime, we can use any language to create a GraphQL schema and build an interface around it.

There are many GraphQL implementations that we can use today to create a GraphQL schema. There are GraphQL implementations for JavaScript, Java, Ruby, Scala, Python, and many more. Some GraphQL implementations will even allow us to ask for data directly from SQL databases.

We will be using the GraphQL JavaScript implementation (`graphql-js`) to create our schema. It's a complete reference implementation that is well-documented and easy to work with. It's also open source and easy to read. It's currently hosted at <https://github.com/graphql/graphql-js>.

In order to use JavaScript on the server side, we will need to install Node.js first. Node.js allows us to execute JavaScript code on the server using the same engine that powers the Chrome browser, V8.

Installing Node.js

To install Node.js, we can download and use the binary installers from <https://nodejs.org/en/>. On Mac and Linux, we can also set up Node.js using **Node Version Manager (NVM)**. NVM targets the task of managing multiple Node.js versions, but it also uses a user-install by default, which keeps everything related to Node.js in one directory. With NVM, it's also easier to update current Node.js installations with newer versions of Node.js when they become available.

NVM does not work on Microsoft Windows, but there are alternatives such as the *nvm-windows* project, and the *nodist* project. The Windows installer available on the Node.js website is also another good option for Microsoft Windows users.

I will be assuming a Linux-based shell environment in the commands and outputs presented in this book. In Microsoft Windows environments, commands and outputs might be slightly different. In all the command lines, the part before the \$ sign will be the working directory where the command was issued, and the part after the \$ sign is the command itself. For example, with the line:

```
~ $ ls
```

This means run the command `ls` in the home directory ~.



To install NVM (on Mac or Linux), we can execute the following command:

```
~ $ curl -o-
https://raw.githubusercontent.com/creationix/nvm/v0.31.4/install.sh | bash
```

The command downloads the `install.sh` script from GitHub (using `curl`), then it executes the script with the `bash` command. The script clones the NVM repository to `~/nvm` and adds a line to the profile file (`~/.bash_profile`, `~/.zshrc` or `~/.profile`) to prepare the NVM environment.



This command uses version 0.31.4 of NVM, which was the latest version at the time of writing. Check the NVM repository on GitHub to see the latest:
<https://github.com/creationix/nvm>.

If we open a new terminal window now, we should have the `nvm` command line available in the environment. To verify:

```
~ $ nvm --version
0.31.4
```

Once we have the `nvm` command available, we can use it to install the latest Node.js:

```
~ $ nvm install node
Downloading https://nodejs.org/dist/v6.3.1/node-v6.3.1-darwin-x64.tar.gz...
Creating default alias: default -> node (-> v6.3.1)
```

This command installs the latest Node available. At the time of writing, it was 6.3.1; you can also instruct `nvm` to install Node v-6.3.1 instead using:

```
~ $ nvm install 6.3.1
```

NVM will make a few Node.js commands available for us. The most important ones are:

- `node`: we use this command to execute a JavaScript file on the server, for example: `node script.js`. If we invoke the `node` command by itself without a target to execute, it runs in **REPL mode (Read, Eval, Print, Loop)**. We can use this mode to test simple JavaScript one-liners while we are in the terminal.
- `npm`: we use this command to install, uninstall, or update a Node.js package.

The GraphQL JavaScript reference implementation is published as an `npm` package under the name `graphql`. We can use `npm` to install it locally for our projects.

Defining the schema

Let's create a new directory for our project:

```
~ $ mkdir ~/graphql-project  
~ $ cd ~/graphql-project
```

We're going to make our project depend on the GraphQL JavaScript library. To document this dependency, we need to create a `package.json` file in our project. A `package.json` file is used by npm to store information about the project, including any dependencies on other packages. We can use the `npm init` command to create the `package.json` file:

```
~/graphql-project $ npm init
```

The `npm init` command will ask a few questions about the project, and it will use our answers to create a `package.json` file. The default answers are usually a good start.

With a `package.json` file created, we can now bring in the `graphql` library using this command:

```
~/graphql-project $ npm install --save graphql  
└── graphql@0.6.2
```

This command will do two things:

- Download the `graphql` library from <https://www.npmjs.com/> and make it available locally under a `node_modules` directory in our project
- Update the `package.json` file to document this new dependency. This is because we used the `--save` option



When committing the changes made so far to source control, remember to ignore this newly created `node_modules` directory. I use Git, and with Git, we can just add a `node_modules` line to a `.gitignore` file. Chapter 8, *Deploying to the Cloud*, has instructions about Git.

With the `graphql` npm package available locally, we can now write our schema. Create a `schema` directory at the root level of our project and create a `main.js` file there. We will define our GraphQL schema object in this `main.js` file:

```
~/graphql-project $ mkdir schema && touch schema/main.js
```

The first thing we need to do in `main.js` is to import a few classes from the `graphql` package:

```
const {  
  GraphQLSchema,  
  GraphQLObjectType,  
  GraphQLString  
} = require('graphql');
```



We will be using the Node.js `require` syntax to import dependencies. The official JavaScript syntax for this task is different, it uses an `import` statement instead, but this official syntax is not supported by Node.js yet (as of 6.3.1). You'll find a lot of examples on the Web that use the `import` syntax, and use Babel to compile it into the `require` syntax which Node.js can work with. In future versions of Node.js, the `import` syntax will be supported natively:

```
// Instead of:  
const graphql = require('graphql');  
// We can do:  
import graphql from 'graphql';
```

We imported only the three helpers that we will be using in the first example, but there are many other helpers we can import from the `graphql` library. We will import the other helpers later when we need them.



The examples demonstrated in this introduction chapter are only to give you a taste of the powerful features in GraphQL. We're only exploring a few features though, and more features will be introduced in later chapters. You will also most likely see syntax that is new to you; the GraphQL language syntax will be covered in detail in Chapter 2, *The Query Language*, and the GraphQL schema syntax will be covered in detail in Chapter 3, *The GraphQL Schema*.

`GraphQLSchema` is the class we can use to instantiate our example schema. Let's name this example schema object `mySchema`:

```
const mySchema = new GraphQLSchema({  
  // root query & root mutation definitions  
});
```

A GraphQL schema can expose multiple capabilities. If we want clients to be able to ask for data, we need to define a query property on the schema. If we want to support any kind of insert, update, or delete operations on our data, we need to define a mutation property on the schema.

The query and mutation properties are instances of the `GraphQLObjectType` class. Let's start with a simple example query. Under the helpers we defined in `schema/main.js`, define a `queryType` object:

```
const queryType = new GraphQLObjectType({
  name: 'RootQuery',
  fields: {
    hello: {
      type: GraphQLString,
      resolve: () => 'world'
    }
  }
});
```

Under the `queryType` definition, update the `mySchema` object to use `queryType` for its query configuration property:

```
const mySchema = new GraphQLSchema({
  query: queryType
});
```

We start by assigning the `queryType` object as the `query` property of the `GraphQLSchema` instance (`mySchema`). `queryType` is an instance of `GraphQLObjectType` and we give it the name `RootQuery`. The name can be anything of course.

The `fields` property on a GraphQL object is where we define the fields that can be used in a GraphQL query to ask about that object. We define an example `hello` field, which is a `GraphQLString`.

Every field in a GraphQL object can define a `resolve()` function. The `resolve()` function is what the `graphql` library executes when it tries to answer queries asking about that field. For our simple example, we resolve that `hello` field with the string `world`.

Finally, to be able to use this simple schema from other parts of our application, we need to export it:

```
module.exports = mySchema;
```

Here's the complete `schema/main.js` file so far which defines and exports the `mySchema` object:

```
const {
  GraphQLSchema,
  GraphQLObjectType,
  GraphQLString
} = require('graphql');

const queryType = new GraphQLObjectType({
  name: 'RootQuery',
  fields: {
    hello: {
      type: GraphQLString,
      resolve: () => 'world'
    }
  }
});

const mySchema = new GraphQLSchema({
  query: queryType
});

module.exports = mySchema;
```

I am going to maintain a GitHub repository for this book. This repo will have incremental commits, branches, and tags associated with chapters and sections of the book. You can clone the repo from <https://github.com/edgecoders/learning-graphql-and-relay> and check your progress against the `git` tag for the chapter's section.

For example, after the *Defining the schema* section here in this chapter, you can `git checkout chapter1-defining-the-schema` to see the code as it is exactly at this point in the chapter.

Every point in the book that has a matching `git` tag in the repo will have a `#GitTag` line referencing that tag. Compare your code to the code in the repo to debug any problems you run into. The repo will also have branches for each chapter; if you want to check out the code as it was at the end of Chapter 4, *Configuring React Applications to Use Relay*, for example, you can `git checkout chapter4`.

If you have comments or questions, feel free to submit a GitHub issue on the repo.

*** #GitTag: chapter1-defining-the-schema ***



Using the schema

Our schema wouldn't be any good unless we could execute queries against it. To do so, we need to create an interface between the user and the schema. This interface will be used to take input from the user (a GraphQL query), and give the user an output (a GraphQL JSON response).

The simplest interface we can use here is a regular Linux command. We can use its standard input (`stdin`) to supply a GraphQL request, and its standard output (`stdout`) to respond with the GraphQL server answer for the requested query.

In the root level of our project, create an `index.js` file with this content:

```
const { graphql } = require('graphql');
const readline = require('readline');

const mySchema = require('./schema/main');

const rli = readline.createInterface({
  input: process.stdin,
  output: process.stdout
});

rli.question('Client Request: ', inputQuery => {
  graphql(mySchema, inputQuery).then(result => {
    console.log('Server Answer :', result.data);
  });
}

rli.close();
});
```



readline is one option of many we can use to interface our GraphQL schema, and it's not a required package to use with GraphQL. We used it here to make for a simple example.

We start by importing the `graphql` library and the `readline` library. We also need to use our GraphQL schema that we defined earlier (`mySchema`), so we import that as well.

We need to create an interface that understands `stdin` and `stdout`, and then just use that interface (`rli`) with a `.question()` function, which takes a string argument and a callback argument.

The `Client Request: string` is what the interface will display as an input prompt when we execute the script. In the second argument, which is a callback, `readline` will expose what the user types as the `inputQuery` variable.

The `graphql()` function we imported on the first line can then be used to execute the user's GraphQL query against our defined schema. If our server receives a valid query that it can understand, it will respond with a promise that resolves to a JSON response for that query. To use the standard `out (stdout)` for our interface, we can use `console.log` to log the promise-resolved JSON result.

To test this `index.js` script, we can execute it with the `node` command:

```
~/graphql-project $ node index.js
Client Request: { hello }
Server Answer : { hello: 'world' }
```

Our script starts by prompting the user to enter input for `Client Request`. We typed the string `{ hello }`. This string represents a client asking about the `hello` root field that we defined in our schema.

The server's answer is a JSON string that matches our request query. It uses the `resolve()` function's returned value (`world`) as the value for the `hello` field:

```
*** #GitTag: chapter1-using-the-schema ***
```

Rolling the dice

Let's add another capability to our GraphQL Server. We will make it simulate a simple two-dice roll.

We'll design our server to be able to respond to the following query:

```
Client Request: { diceRoll }
```

We expect the server to reply with two random values between 1 and 6:

```
Server Answer : { diceRoll: [2, 5] }
```

The data type of our planned response is an array. In the `graphql` library, we use a `GraphQLList` to represent an array type, and we will use a `GraphQLInt` type to represent the elements of our random integers array. `GraphQLList` and `GraphQLInt` are both helpers available from the `graphql` library.

Update the require line in `schema/main.js` and add the two new helpers:

```
const {
  // ...
  GraphQLInt,
  GraphQLList
} = require('graphql');
```

We need a function that can respond with a random number between 1 and 6. To accomplish that, we can use a combination of `Math.random` and `Math.floor`. Add the following function right after the require line in `schema/main.js`:

```
const roll = () => Math.floor(6 * Math.random()) + 1;
```

In the `mySchema` object, we need to add a new root field, `diceRoll`, to our list of fields and make its type a `GraphQLList` of `GraphQLInt`:

```
fields: {
  // The hello field definition...
  diceRoll: {
    type: new GraphQLList(GraphQLInt),
    resolve: () => [roll(), roll()]
  }
}
```

Our `diceRoll` field resolves with an array that has two integers, which are both computed from a `roll()` function call. The type for our resolved value here is a `GraphQLList`, and each item in that list is a `GraphQLInt`.

To test this new capability, we will execute the same `babel-node` as previously command.

Here are three different dice rolls to make sure we are getting random values:

```
~/graphql-project $ node index.js
Client Request: { diceRoll }
Server Answer : { diceRoll: [ 3, 6 ] }

~/graphql-project $ node index.js
Client Request: { diceRoll }
Server Answer : { diceRoll: [ 6, 6 ] }

~/graphql-project $ node index.js
Client Request: { diceRoll }
Server Answer : { diceRoll: [ 3, 2 ] }

*** #GitTag: chapter1-rolling-the-dice ***
```

Using field arguments

We want to make our server capable of rolling more than two dice. In fact, we want the clients to be able to customize their question and tell us how many dice to roll.

We can utilize a GraphQL field argument for that purpose. The client can ask a query, such as:

```
{ diceRoll(count: 5) }
```

For the above query, the client expects the server to roll five dice. Here's how we can change the `diceRoll` field definition to allow for that operation on the server:

```
diceRoll: {
  type: new GraphQLList(GraphQLInt),
  args: {
    count: { type: GraphQLInt }
  },
  resolve: (_, args) => {
    let rolls = [];
    for (let i = 0; i < args.count; i++) {
      rolls.push(roll());
    }
    return rolls;
  }
}
```

To make our GraphQL server aware of the arguments passed to a field, we define an `args` property on that field with the name and type of the allowed argument. For our example, `count` is a `GraphQLInt`.

We can use the same `roll()` function to generate one random value, but since we're now receiving a `count` from the user, we need to dynamically construct an array using that `count` value. Inside the `resolve()` function, all the arguments that get passed to that field in a client request can be accessed using the `second` argument for the `resolve()` function itself. We can read the value the user enters for the `count` argument, using `args.count`.

In a simple `for` loop, we construct an array of `args.count` length, and fill it with a different `roll()` call for each element.

We named the first argument for `resolve()` with an underscore because we're not using it for this example. This argument represents the parent object and it's undefined on the first root-level queries. We'll learn about arguments for the resolver functions in [Chapter 3, The GraphQL Schema](#).

To test this new feature:

```
~/graphql-project $ node index.js

Client Request: { diceRoll(count: 5) }
Server Answer : { diceRoll: [ 6, 5, 3, 5, 1 ] }
```



Note that this change makes the code depend on the `count` argument. If we want the code to work with or without a `count` argument, we should add a default value for that argument. In GraphQL, we can define a default value for any argument using the optional `defaultValue` property:

```
args: {
  count: {
    type: GraphQLInt,
    defaultValue: 2
  }
}

*** #GitTag: chapter1-using-field-arguments ***
```

Setting up MongoDB

An API is nothing without access to a database. Let's set up a local MongoDB instance, add some data in there, and make sure we can access that data through our GraphQL schema.

MongoDB can be locally installed on multiple platforms. Check the documentation site for instructions for your platform (<https://docs.mongodb.com/manual/installation/>).

For Mac, the easiest way is probably Homebrew:

```
~ $ brew install mongodb
```

Create a `db` folder inside a `data` folder. The default location is `/data/db`:

```
~ $ sudo mkdir -p /data/db
```

Change the owner of the `/data` folder to be the current logged-in user:

```
~ $ sudo chown -R $USER /data
```

Start the MongoDB server:

```
~ $ mongod
```



In Chapter 8, *Deploying to the Cloud*, we go over instructions to install MongoDB on an Ubuntu machine.

If everything worked correctly, we should be able to open a new terminal and test the mongo CLI:

```
~/graphql-project $ mongo  
  
MongoDB shell version: 3.2.8  
connecting to: test  
> db.getName()  
test  
>
```



We're using MongoDB version 3.2.8 here. Make sure that you have this version or newer versions of MongoDB.

Let's go ahead and create a new collection to hold some test data. Let's name that collection users:

```
> db.createCollection("users")  
{ "ok" : 1 }
```

Now we can use the users collection to add documents that represent users. We can use the MongoDB insertOne() function for that:

```
> db.users.insertOne({  
    firstName: "John",  
    lastName: "Doe",  
})
```

We should see an output like this:

```
{  
    "acknowledged" : true,  
    "insertedId" : ObjectId("56e729d36d87ae04333aa4e1")  
}
```

Let's go ahead and add another user:

```
> db.users.insertOne({  
  firstName: "Jane",  
  lastName: "Doe",  
})
```

We can now verify that we have two user documents in the `users` collection:

```
> db.users.count()  
2
```



MongoDB has a built-in unique object ID which you can see in the output for `insertOne()`. We will be using this object ID later in the book.

Now that we have a running MongoDB, and we have some test data in there, it's time to see how we can read this data using a GraphQL API.

To communicate with a MongoDB from a Node.js application, we need to install a *driver*. There are many options that we can choose from, but GraphQL requires a driver that supports promises. We will use the official MongoDB Node.js driver, which supports promises. Instructions on how to install and run the driver can be found at <https://docs.mongodb.com/ecosystem/drivers/node-js/>.

To install the MongoDB official Node.js driver under our `graphql-project` app, we do this:

```
~/graphql-project $ npm install --save mongodb  
└── mongodb@2.2.5
```

We can now use this `mongodb` npm package to connect to our local MongoDB server from within our Node application. In `index.js`, add the following:

```
const { MongoClient } = require('mongodb');  
const assert = require('assert');  
  
const MONGO_URL = 'mongodb://localhost:27017/test';  
  
MongoClient.connect(MONGO_URL, (err, db) => {  
  assert.equal(null, err);  
  console.log('Connected to MongoDB server');  
  
  // The readline interface code  
});
```



The MONGO_URL variable value should not be hardcoded in code like this. Instead, we can use a Node process environment variable to set it to a certain value before executing the code. On a production machine, we would be able to use the same code and set the process environment variable to a different value.

Use the `export` command to set the environment variable value:

```
export MONGO_URL=mongodb://localhost:27017/test Then in the Node code, we can read the exported value by using this:  
process.env.MONGO_URL
```

If we now execute the `node index.js` command, we should see the `Connected to MongoDB server` line right before we ask for the Client Request.



At this point, the Node.js process will not exit after our interaction with it. We'll need to force exit the process with `Ctrl + C` to restart it.

Let's start our database API with a simple field that can answer this question: how many total users do we have in the database?

The query could be something like this:

```
{ usersCount }
```

To be able to use a MongoDB driver call inside our schema `main.js` file, we need access to the `db` object that the `MongoClient.connect()` function exposed for us in its callback. We can use the `db` object to count the user documents by simply running the promise:

```
db.collection('users').count()  
.then(usersCount => console.log(usersCount));
```

Since we only have access to the `db` object in `index.js` within the `connect()` function's callback, we need to pass a reference to that `db` object to our `graphql()` function. We can do that using the fourth argument for the `graphql()` function, which accepts a `contextValue` object of `globals`, and the GraphQL engine will pass this context object to all the resolver functions as their third argument. Modify the `graphql` function call within the `readline` interface in `index.js` to be:

```
graphql(mySchema, inputQuery, {}, { db }).then(result => {  
  console.log('Server Answer :', result.data);  
  db.close(() => rli.close());  
});
```



The third argument to the `graphql()` function is called the `rootValue`, which gets passed as the first argument to the resolver function on the top level type. We are not using that feature here.

We passed the connected database object `db` as part of the global context object. This will enable us to use `db` within any resolver function.

Note also how we're now closing the `rli` interface within the callback for the operation that closes the `db`. We should not leave any open `db` connections behind.

Here's how we can now use the resolver third argument to resolve our `usersCount` top-level field with the `dbcount()` operation:

```
fields: {
  // "hello" and "diceRoll"...
  usersCount: {
    type: GraphQLInt,
    resolve: (_, args, { db }) =>
      db.collection('users').count()
  }
}
```

A couple of things to notice about this code:

- We destructured the `db` object from the third argument for the `resolve()` function so that we can use it directly (instead of `context.db`).
- We returned the promise itself from the `resolve()` function. The GraphQL executor has native support for promises. Any `resolve()` function that returns a promise will be handled by the executor itself. The executor will either successfully resolve the promise and then resolve the query field with the promise-resolved value, or it will reject the promise and return an error to the user.

We can test our query now:

```
~/graphql-project $ node index.js
Connected to MongoDB server
Client Request: { usersCount }
Server Answer : { usersCount: 2 }

*** #GitTag: chapter1-setting-up-mongodb ***
```

Setting up an HTTP interface

Let's now see how we can use the `graphql()` function under another interface, an HTTP one.

We want our users to be able to send us a GraphQL request via HTTP. For example, to ask for the same `usersCount` field, we want the users to do something like this:

```
/graphql?query={usersCount}
```

We can use the `Express.js` node framework to handle and parse HTTP requests, and within an `Express.js` route, we can use the `graphql()` function, for example:

```
const app = express();

app.use('/graphql', (req, res) => {
  // use graphql() to respond with JSON objects
});
```

However, instead of manually handling the `req/res` objects, there is a GraphQL `Express.js` middleware that we can use, `express-graphql`. This middleware wraps the `graphql()` function and prepares it to be used by `Express.js` directly. Let's go ahead and bring in both the `Express.js` library and this middleware:

```
~/graphql-project $ npm install --save express express-graphql
├── express@4.14.0
└── express-graphql@0.5.3
```

In `index.js`, we can now import both `express` and the `express-graphql` middleware:

```
const graphqlHTTP = require('express-graphql');
const express = require('express');

const app = express();
```

With these imports, the middleware main function will now be available as `graphqlHTTP()`. We can now use it in an Express route handler. Inside the `MongoClient.connect()` callback, we can do this:

```
app.use('/graphql', graphqlHTTP({
  schema: mySchema,
  context: { db }
}));

app.listen(3000, () =>
  console.log('Running Express.js on port 3000')
```

```
);
```



Note that at this point, we can remove the `readline` interface code as we are no longer using it. Our GraphQL interface from now on will be an HTTP endpoint.

The `app.use` line defines a route at `/graphql` and delegates the handling of that route to the `express-graphql` middleware that we imported. We pass two objects to the middleware, the `mySchema` object and the `context` object. We're not passing any input query here because this code just prepares the HTTP endpoint, and we will be able to read the input query directly from a URL field.



The `app.listen()` function is the call we need to start our `Express.js` app. Its first argument is the port to use, and its second argument is a callback we can use after `Express.js` has started.

We can now test our HTTP-mounted GraphQL executor with:

```
~/graphql-project $ node index.js
```

```
Connected to MongoDB server
Running Express.js on port 3000
```

In a browser window, go to:

```
http://localhost:3000/graphql?query={usersCount}
```

A screenshot of a web browser window. The address bar shows "localhost:3000/graphql?query={usersCount}" and the page content area displays the following JSON response:

```
{
  - data: {
      usersCount: 2
    }
}
```

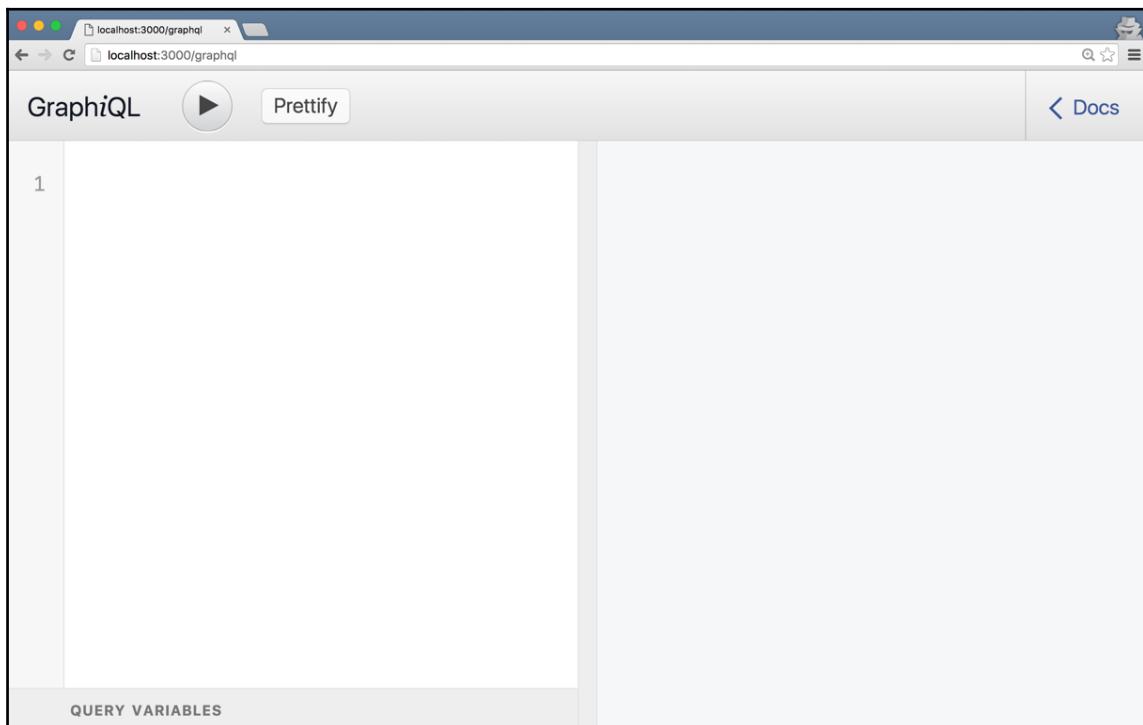
```
*** GitTag: chapter1-setting-up-an-http-interface ***
```

The GraphiQL editor

The `graphqlHTTP()` middleware function accepts another property on its parameter object `graphiql`; let's set it to `true`:

```
app.use('/graphql', graphqlHTTP({
  schema: mySchema,
  context: { db },
  graphiql: true
}));
```

When we restart the server now and navigate to `http://localhost:3000/graphiql`, we'll get an instance of the GraphiQL editor running locally on our GraphQL schema:



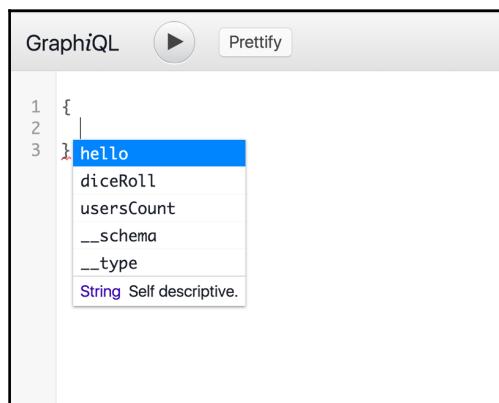
GraphiQL is an interactive playground where we can explore our GraphQL queries and mutations before we officially use them. GraphiQL is written in React and GraphQL, and it runs completely within the browser.

GraphQL has many powerful editor features such as syntax highlighting, code folding, and error highlighting and reporting. Thanks to GraphQL's introspective nature, GraphQL also has intelligent type-ahead for fields, arguments, and types.

Put the cursor in the left editor area, and type a selection set:

```
{  
}
```

Place the cursor inside that selection set and press *Ctrl + space*. You should see a list of all fields that our GraphQL schema supports, which are the three fields that we have defined so far (`hello`, `diceRoll`, and `usersCount`):



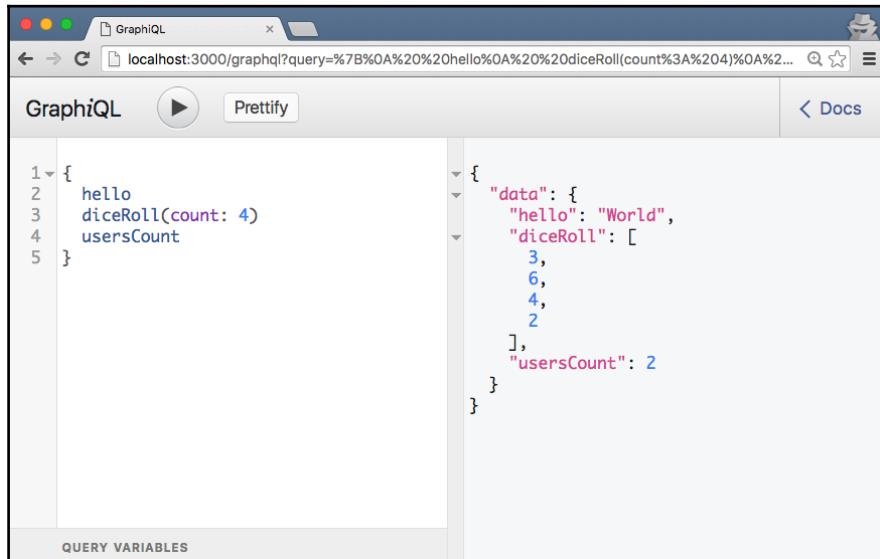
If *Ctrl + space* does not work, try *Cmd + space*, *Alt + space*, or *Shift + space*.



The `__schema` and `__type` fields can be used to introspectively query the GraphQL schema about what fields and types it supports. We will talk about GraphQL introspection in Chapter 3, *The GraphQL Schema*.

When we start typing, this list starts to get filtered accordingly. The list respects the context of the cursor; for example, if we place the cursor inside the arguments of `diceRoll()`, we'll get the only argument we defined for `diceRoll`, the `count` argument.

Go ahead and read all the root fields that our schema support, and see how the data gets reported on the right side with a formatted JSON object:



The screenshot shows the GraphiQL interface running in a browser window. The title bar says "GraphQL". The URL bar shows "localhost:3000/graphql?query=%7B%0A%20%20hello%0A%20%20diceRoll(count%3A%204)%0A%20%20usersCount%0A%7D". The main area has two panes. The left pane contains the GraphQL query:

```
1 {  
2   hello  
3   diceRoll(count: 4)  
4   usersCount  
5 }
```

The right pane displays the resulting JSON data:

```
{  
  "data": {  
    "hello": "World",  
    "diceRoll": [  
      3,  
      6,  
      4,  
      2  
    ],  
    "usersCount": 2  
  }  
}
```

Below the main area, there is a "QUERY VARIABLES" section.

*** GitTag: chapter1-the-graphiql-editor ***

Summary

In this chapter, we introduced GraphQL and Relay and talked about what they are, what problems they solve, and why they are needed. We saw examples of GraphQL queries and how they get used in Relay. We talked about RESTful APIs, and how they compare to GraphQL APIs. Using an example user interface, we learned how a simple GraphQL API makes a big difference in how we communicate with a data service.

We talked about Relay, its core principles, how it handles storage and caching, and how it uniquely identifies every object in an application. We also talked about Relay's connection model for pagination.

We learned how to create a simple GraphQL schema, and how to use it with different interfaces, including an HTTP interface. We also learned how to use GraphQL to read data from MongoDB. We explored the GraphiQL editor and used it to inspect our GraphQL schema and read the data it exposes.

2

The Query Language

The GraphQL query language is designed around flexible syntax that's easy to read and understand. In this chapter, we'll explore the language syntax, and learn the different features it supports. We will cover the following topics:

- Documents and operations
- Fields
- Variables
- Directives
- Aliases
- Fragments
- Mutations

Documents and operations

To communicate with a GraphQL service, we send it a text *document* written in the GraphQL query language. A GraphQL document contains one or more *operations*, and these operations can be either read or write operations. We use queries for read operations, and mutations for write operations.



GraphQL will be supporting more operations in the future. For example, there will soon be a subscription operation for real-time data processing. The support for the subscription operation is already in the JavaScript implementation but it's still in an early experimental phase.

Here's a query to read the list of comments on a blog article:

```
# Find one article and its list of comments:  
query ArticleComments {  
  article(articleId: 42) {  
    comments {  
      commentId  
      formattedBody  
      timestamp  
    }  
  }  
}
```

Notice how we used the keyword `query` before the main selection set and we gave this query an identifying name `ArticleComments`. We've been using the *query shorthand* in Chapter 1, *An Introduction to GraphQL and Relay*, and it didn't include a type for the operation or a name for it. By default, the type of operation is a query. If there is only one simple operation in the document, the name can be omitted. The query name can be anything and it's optional for a single query. However, it's a good habit to always give GraphQL operations descriptive names.

A GraphQL document may contain multiple operations, in which case the name of the desired operation to be executed must be provided. In addition to queries and mutations, a GraphQL document can also contain fragments. We can think of fragments as partial operations that are generic and reusable.

Most of the query language elements are expressed using ASCII characters, but Unicode characters are acceptable inside string values and comments. The first line in the previous example is a comment, any text that appears between a `#` marker and a line terminator is a comment and it will be ignored by the parser. The GraphQL parser will also ignore the following tokens unless they appear within a string value: line terminators, commas, and most whitespace characters. These optional tokens are used to make the source text of a GraphQL document more readable and should be used where possible. The `ArticleComments` query would be hard to read if it was just one big single line:

```
{article(articleId:42){comments{commentId formattedBody timestamp}}}
```

We can execute the previous line normally, but this style of query should be avoided.

Fields

In the ArticleComments query, we're asking the server about the article with articleId #42. For that article, we're asking the server for its list of comments, and for every comment, we're asking the server about the commentId, its formattedBody, and its timestamp.

Here is a possible response to the ArticleComments query:

```
{  
  "article": {  
    "comments": [  
      {  
        "commentId": 1,  
        "formattedBody": "GraphQL is <strong>cool</strong>",  
        "timestamp": "12/12/2015 - 15:15"  
      },  
      {  
        "commentId": 2,  
        "formattedBody": "What's wrong with <em>REST</em>!",  
        "timestamp": "12/12/2015 - 15:25"  
      }  
    ]  
  }  
}
```

This response has sections that represent the different fields in our GraphQL query. In our query, article is called a *field*, and so are comments, commentId, formattedBody, and timestamp. A field can be mapped to either a primitive value in the response, such as the text representing the formattedBody, or to an object or array of objects in the response.

Here are the different mappings we have in the ArticleComments query:

- The article field in the query maps to a single object in the response
- The comments field in the query maps to an array of objects in the response
- The commentId field in the query maps to a scalar integer value
- The formattedBody and timestamp fields in the query both map to a scalar string value

You can think of fields as functions; they return something in the response. They also take arguments, for example the `article` field takes an integer argument `articleId`. On the server side, we can use field arguments to customize the response to be resolved by the field. In this case, the server customized the response to return the data associated with article #42.

Fields in a GraphQL query map to properties on objects. The `comments` field, for example, is a property on an article object, and `commentId`, `formattedBody`, and `timestamps` are all properties on a comment object. The `article` field is also a property on what is called the *root query object*. A root query object is an entry point; one of the possible many points on the graph that we can start with in our queries.

The curly braces in the GraphQL query are called *selection sets*, and those are nestable; one selection set can contain other selection sets. When we ask for a field that maps to an object or an array of objects, we need a new selection set to tell the GraphQL server which scalar properties we'd like to read from those objects. The innermost selection set of a GraphQL query should always contain fields that resolve to scalar values. In GraphQL, we call fields that map to objects or an array of objects *complex fields*. If we try to query for a complex field such as `article` without a selection set, the GraphQL error will be clear: `Field "article" of type "Article" must have a sub selection.`

Variables

The `ArticleComments` query works for a single article, and that's why we needed to pass an `articleId` for it. However, we hardcoded the value of `articleId` in the query string itself. That makes the query not usable for other articles. To make the query reusable, we need to make it generic by using a GraphQL *variable* as the input for `article`:

```
query ArticleComments($articleId: Int!) {
  article(articleId: $articleId) {
    comments {
      commentId
      formattedBody
      timestamp
    }
  }
}
```

Notice how we first define the variable at the top of our query operation (`$articleId: Int!`). This sets the scope of the variable `$articleId` so that we can use it anywhere inside our query operation. The type of the `$articleId` variable is `Int`; the trailing exclamation mark after the type indicates that this variable is required and can't be null.

To execute the generic query, we supply a JSON object for the `variables` input which we pass to our GraphQL query executor along with the `query` input. For example, to execute the query with the value `42` for the variable `$articleId`, we can send this value for `variables`:

```
{  
  "articleId": 42  
}
```

For an HTTP interface, our operation request can now be sent using:

```
/graphql?query={...}&variables={...}
```

Using variables allow the clients to avoid any string building operation at runtime. The `42` value will most likely come from a different source, such as the location in the URL. Instead of building a query string using string concatenation for that variable, we can just pass the variable to the executor along with the generic reusable query.

Variables have to be unique in a single operation, but we can use the same variable name in different operations. If we define a variable for an operation, that variable has to be used at least once in that operation. GraphQL will reply with an error if a variable was defined but never used.

In GraphiQL, we can use the lower-left corner (labeled *QUERY VARIABLES*) to define the values object for all variables in the document:

The screenshot shows the GraphiQL interface. In the top left, there's a code editor with a query:1 query PersonInfo(\$personId: ID) {
2 person(personID: \$personId) {
3 name
4 }
5 }
6A "Prettify" button is next to it. On the right, the results are displayed as JSON:{
 "data": {
 "person": {
 "name": "Darth Vader"
 }
 }
}Below the code editor is a "QUERY VARIABLES" section containing:1 {
2 "personId": 4
3 }

Directives

Sometimes, using field arguments to customize the behavior of the GraphQL server execution engine is not enough. For example, what if we had a variable in our application, and we wanted the GraphQL server to customize the structure of the response based on this variable? When the variable is true, we want certain fields to be omitted from the response.

We can provide options to alter the GraphQL runtime execution using directives. Directives have three characteristics:

- A unique name to identify them.
- A list of arguments, just like fields. Arguments may accept values of any input type.
- A list of locations where the use of the directive is accepted. Directives can be used in multiple locations in a GraphQL document, including an operation, a fragment, or a field. Each directive defines its list of acceptable locations.

There are two main built-in directives that should be supported by a GraphQL server:

- `@include`, which accepts a Boolean `if` argument, and directs the GraphQL executor to include a field or a fragment only when the `if` argument is true:

```
field @include(if: $BooleanValue)
```

- `@skip`, which accepts a Boolean `if` argument, and directs the GraphQL executor to skip a field or fragment when the `if` argument is true:

```
field @skip(if: $BooleanValue)
```



GraphQL is likely to have a lot more built-in directives in the future. For example, there might soon be a `@deprecated` directive for marking elements no longer supported, an `@export` directive to export a field in one query as a dependency for another query during a batch operation, a `@defer` directive for delaying the response of a certain field, and a `@stream` directive to convert a field response into a real-time stream. Most of these new directives are still experimental as of this writing.

Directives are commonly used with variables to customize the response based on variables' values. For example, in our `ArticleComments` query, we can have a variable `$showEmails` that can be either true or false. We want the GraphQL server to only respond with e-mail information when `$showEmails` is set to true. When `$showEmails` is false, we want the server to respond with the author's website address instead:

```
query ArticleComments($articleId: Int!, $showEmails: Boolean!) {
  article(articleId: $articleId) {
    comments {
      commentId
      formattedBody
      timestamp
      author {
        name
        email @include(if: $showEmails)
        website @skip(if: $showEmails)
      }
    }
  }
}
```

When we execute this query with `$showEmails` set to true (in the variables section), the GraphQL server will include the author's e-mail and it will not include the author's website. When we execute the query with `$showEmails` set to false, the server will include the author's website but not the author's e-mail.

We can use directives with complex fields as well. For example, to control the author section itself on every comment, we can do this:

```
query ArticleComments($articleId: Int!, $showAuthor: Boolean!) {
  article(articleId: $articleId) {
    comments {
      commentId
      formattedBody
      timestamp
      author @include(if: $showAuthor) {
        name
      }
    }
  }
}
```

Aliases

Sometimes, the data exposed by the server might have different property names than what the UI is using. For example, let's assume our UI represents every article as `post` and represents the list of comments on an article as `responses`, and for every response, the UI uses a `responseId` instead of a `commentId`.

Here's the ideal JSON object that this UI can consume:

```
{
  "post": {
    "responses": [
      {
        "responseId": 1,
        "formattedBody": "GraphQL is <strong>cool</strong>",
        "timestamp": "12/12/2015 - 15:15"
      },
      {
        "responseId": 2,
        "formattedBody": "What's wrong with <em>REST</em>!",
        "timestamp": "12/12/2015 - 15:25"
      }
    ]
  }
}
```

We have the same data here but with slightly different property names. Instead of processing the original response on the client side and copying it to a new structure suitable for our UI, we can use GraphQL field aliases to instruct the GraphQL server to respond with its data using different property names. We can use aliases on any field to customize its appearance in the response:

```
query ArticleResponses {
  post: article(articleId: 42) {
    responses: comments {
      responseId: commentId
      formattedBody
      timestamp
    }
  }
}
```

The `ArticleResponses` query is asking for the same data but it uses aliases to rename some fields in the response. The server will respond with the exact JSON that the UI wants to use.

We can also use aliases to ask for the same field multiple times:

```
query TwoArticles {
  firstArticle: article(articleId: 42) {
    comments {
      commentId
      formattedBody
      timestamp
    }
  }
  secondArticle: article(articleId: 43) {
    comments {
      commentId
      formattedBody
      timestamp
    }
  }
}
```

The aliases a client specifies in a GraphQL query get used in the server response. Using this feature, clients have an extra level of control over the response, and they don't need to do any extra processing on received data before they can use it.

Fragments

In the last example, we repeated the `comments` section in the `TwoArticles` query twice, once for every article. If we later decide to ask for an extra field on every comment object, we will have to change two places in our query, which is not ideal.

We can use GraphQL *fragments* to refactor this repetition and compose our main query using a smaller query fragment that represents the `comments` section and its fields:

```
query TwoArticles {
  firstArticle: article(articleId: 42) {
    ...CommentList
  }
  secondArticle: article(articleId: 43) {
    ...CommentList
  }
}
fragment CommentList on Article {
  comments {
    commentId
    formattedBody
    timestamp
  }
}
```

We were able to extract the repeated information and represent it with the `CommentList` fragment. A fragment is just a partial operation; we can't use it on its own, but we can use it and reuse it inside a full operation. To use a fragment in an operation, we prefix it with *three dots*.

The three dots operator is known as the spread operator. When a GraphQL server sees three dots followed by a name anywhere in a GraphQL query, it will look for a fragment defined using that same name, and it will spread the content of the fragment in place of its three-dotted name in the query. The content of the fragment has to fit in the place where it is used. That's why our fragment, which was defined on an article object, can only be used within the selection set that expands an article object.

Query fragments usually map to the components of a UI. A possible UI to represent this example's data might have an `Article` component, and that component might contain another component that represents a `CommentList` on that article. Our `CommentList` fragment can be used to represent the data requirement for the `CommentList` UI component. This allows for the isolation of the data every sub-component in our UI is asking for, and it also allows for multiple views to use the same fragment without any duplication of logic.

We can use variables in fragments. When a fragment gets used by an operation, it gets access to the variables defined by that operation.

Here's our `$showAuthor` variable used in the fragment we defined for `CommentList`:

```
query TwoArticles($showAuthor: Boolean!) {
  firstArticle: article(articleId: 42) {
    ...CommentList
  }
  secondArticle: article(articleId: 43) {
    ...CommentList
  }
}
fragment CommentList on Article {
  comments {
    commentId
    formattedBody
    timestamp
    author @include(if: $showAuthor) {
      name
    }
  }
}
```

If the fragment uses a variable, that variable has to be defined by any operation that uses the fragment. Using that fragment in an operation that does not define a matching variable name will result in an error.

We can also use fragments directly inline without giving them a name:

```
query ArticleOrComment {
  node(nodeId: 42) {
    formattedBody
    timestamp
    ... on Article {
      nodeId: articleId
    }
    ... on Comment {
```

```
        nodeId: commentId
    }
},
}
```

Inline fragments are useful inside a type that implements multiple objects, such as the `node` field in the `ArticleOrComment` query. The `node` field is part of Relay's features and it can represent any object in the GraphQL schema. In this query, we're assuming that a node can either be an article or a comment. When the node is an article, we want to read the `articleId` and use it as the `nodeId`, and when it's a comment, we want `commentId` to be the `nodeId`.

Inline fragments can also be used to apply a directive to a group of fields. In this format, we can omit the `on Type` section and assume it matches the enclosing context.

For example, given that an article object has `views` and `likes` properties, we can conditionally include them using the `$showStats` variable like this:

```
query Article($showStats: Boolean!) {
  article(articleId: 42) {
    title
    formattedBody
    ... @include(if: $showStats) {
      views
      likes
    }
  }
}
```

Mutations

Reading is just one of the four CRUD operations that a client can communicate to a server. Most clients will also communicate their need to update the data. With GraphQL, this can be done with **mutations**.

A GraphQL mutation is very similar to a GraphQL query, but with runtime awareness that resolving the mutation will have side effects on some elements of the data. A good GraphQL runtime implementation executes multiple GraphQL mutations in a single request in sequence one by one, while it executes multiple GraphQL queries in the same request in parallel.

GraphQL fields, which we use in both queries and mutations, accept arguments. For mutations, we can use field arguments as data input. Here's an example of a GraphQL mutation that can be used to add a comment to an article using markdown:

```
mutation AddNewComment {  
  addComment(  
    articleId: 42,  
    authorEmail: "mark@fb.com",  
    markdown: "GraphQL is clearly a **game changer**"  
  ) {  
    id,  
    formattedBody  
    timestamp  
  }  
}
```

The markdown feature demonstrates how a GraphQL mutation can handle both writing and reading at the same time. It's just another function that gets resolved on the server, but it will do multiple things. The `addComment` function on the server will first persist the comment data that we receive through field arguments, and it will then read the database-generated timestamp, process the markdown of the comment, and return a JSON object ready to be used by the UI to display that newly created comment. We will see an example of how to define a GraphQL mutation on the server in the next chapter.

Summary

In this chapter, we learned the flexible syntax of the GraphQL query language. We've talked about a GraphQL document and the operations it supports, we've seen examples of how to query for fields, and how to use field arguments to customize the behavior of the GraphQL executor. To further customize the behavior of the executor, we've seen how to use directives and aliases. We learned about using fragments to remove duplication in queries, and we've seen how a GraphQL mutation can be used for a write operation.

In the next chapter, we'll define a GraphQL schema and see how all the operations and features we explored here can be implemented on the server side.

3

The GraphQL Schema

A GraphQL schema is what we write to represent the capabilities of a GraphQL server. In a GraphQL schema, we define the types and directives that we want the server to support.

In this chapter, we will explore working with a GraphQL schema and the core features of a GraphQL runtime. We'll cover the following topics:

- The schema object and the operations it supports
- The GraphQL introspective API
- The GraphQL type system
- Scalar and object types
- Interfaces and unions
- Enums
- Type modifiers
- The resolve function and how to use it with promises in queries and mutations
- Validation rules for the GraphQL executor
- API versioning in GraphQL

The schema object

Let's take a look at the exampleschema from Chapter 1, *An Introduction to GraphQL and Relay*:

```
const queryType = new GraphQLObjectType({  
  name: 'RootQuery',  
  fields: {  
    hello: {  
      // ...  
    }  
  }  
})
```

```
        },
        diceRoll: {
          // ...
        },
        usersCount: {
          // ...
        }
      }
    });
}

const mySchema = new GraphQLSchema({
  query: queryType
});
```

A GraphQL schema can be defined as an instance of the `GraphQLSchema` class. The schema is a representation of the capabilities of a GraphQL server starting from the root fields. In our example, those root fields were `hello`, `diceRoll`, and `usersCount`.

The `mySchema` constant is what we used in [Chapter 1, An Introduction to GraphQL and Relay](#), to build an interface for the user. We can define multiple schemas and build multiple interfaces for them. Every schema is an instance of the `GraphQLSchema` class.

To explain the syntax of the GraphQL JavaScript implementation type system, we'll use the same notations used in the official GraphQL documentation site: <http://graphql.org/>.

This documentation site is for the JavaScript implementation of GraphQL and it uses **FlowType** notations to document the API reference. FlowType is a static type checker for JavaScript that's also used on Facebook.

To read more details about any classes or objects that are introduced here, you can look them up in the <http://graphql.org/> site. The documentation for FlowType is available at <https://flowtype.org/>. Please note that FlowType is *not* required to use GraphQL at all, it's just used in the documentation site to make types and requirements clearer to the reader. The actual JavaScript examples in this book are not using FlowType.



The constructor of the `GraphQLSchema` class expects a configuration object:

```
class GraphQLSchema {
  constructor(config: GraphQLSchemaConfig)
}
```

In a JavaScript class (and in many other languages as well), the `constructor` function is what will be executed when we create a new instance of that class (in JavaScript, we create a new instance using the `new` keyword). According to the previous definition of the `GraphQLSchema` class, the argument we pass to an instance of a `GraphQLSchema` is a `config` object. This `config` object can have one or two properties:

```
type GraphQLSchemaConfig = {
  query: GraphQLObjectType;
  mutation?: ?GraphQLObjectType;
};
```

Here's how we can read the previous notation: a `GraphQLSchema` configuration can be constructed as an object that has one or two properties, a `query` property, and an optional `mutation` property. The values for both the `query` and `mutation` properties have the type `GraphQLObjectType`.



The extra characters you'll see in the official GraphQL API reference documentation (such as the `?` after `mutation` and the `?` before `GraphQLObjectType`) are from the FlowType notations. A `?` after an object property marks that property as optional, and a `?` before a type marks the type with the possibility of a null value.

The `query` and `mutation` properties in a `GraphQLSchema` configuration object represent the root type of their operations. A `query` operation represents a read-only fetch of information, while a `mutation` operation represents a write followed by a read fetch of information.

We define these two operations using the `GraphQLObjectType` class. When we query our GraphQL server, we can start with the fields that are defined on the root `query` `GraphQLObjectType`. The constructor of the `GraphQLObjectType` expects another configuration object:

```
class GraphQLObjectType {
  constructor(config: GraphQLObjectTypeConfig)
}

type GraphQLObjectTypeConfig = {
  name: string;
  description?: ?string;
  fields: GraphQLFieldConfigMapThunk | GraphQLFieldConfigMap;
  interfaces?: ...;
};
```

We will talk about interfaces in a later section. A `GraphQLObjectType` configuration object requires a name, which is just a string, that we can use to identify the object instance. We will see the `name` property on many other objects in a GraphQL document, such as operations, fields, arguments, directives, fragments, and variables. Names in GraphQL are case-sensitive, and they are limited to the ASCII alphanumeric characters plus underscores.

We can also supply an optional `description` property for any object. A description is another string that can be used to add documentation about any GraphQL object. While other languages have separate techniques to write documentation, in GraphQL the documentation is part of the definition of objects and is written using the same implementation language.

The reference notation for the `GraphQLObjectType` configuration object defines the `fields` property to have a type of either `GraphQLFieldConfigMapThunk` or `GraphQLFieldConfigMap`. An `AbcThunk` is just a function that returns `Abc`, so we can define the `fields` property with either an object that directly represents a `GraphQLFieldConfigMap` or with a function that returns a `GraphQLFieldConfigMap`.

```
type GraphQLFieldConfigMapThunk = () => GraphQLFieldConfigMap;
```

We can use the thunk function expression syntax when two types need to refer to each other, or when a type needs to refer to itself in a field.

This latter case happens when there is a self-referential relation. For example, the boss of an employee is also an employee:

```
const EmployeeType = new GraphQLObjectType({
  name: 'Employee',
  fields: () => ({
    name: { type: GraphQLString },
    boss: { type: EmployeeType },
  })
});
```

We have to use a function expression in this case, otherwise `EmployeeType` would not be defined when we use it for the `boss` field.

A GraphQL field configuration map is an object that holds a list of fields, and every field is a configuration object:

```
type GraphQLFieldConfigMap = {
  [fieldName: string]: GraphQLFieldConfig;
};
```

A field configuration object is a simple one; we have already seen most of its properties:

```
type GraphQLFieldConfig = {
  type: GraphQLOutputType;
  description?: ?string;
  args?: GraphQLFieldConfigArgumentMap;
  resolve?: GraphQLFieldResolveFn;
  deprecationReason?: string;
};
```

Each property on this configuration object will lead our discussion about a major GraphQL core runtime concept. Let's start by talking about the `description` property and its role in the GraphQL introspective nature.

Introspection

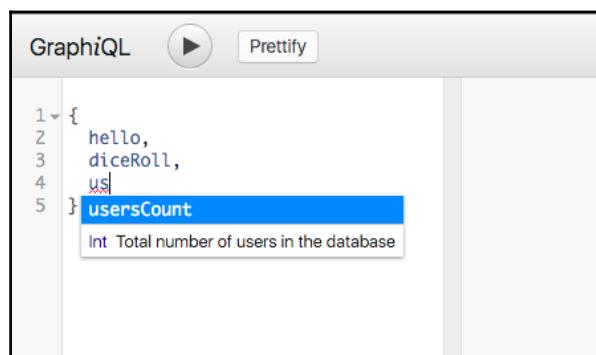
The `description` property, which can be defined on many GraphQL schema elements, is used to give clients some details about an object. When clients read the schema information, the description of each object will be available to them.

For example, GraphQL shows the `description` value for a field object in the type-ahead dropdown that lists fields available inside a selection set. To see that in action, let's give our example schema fields some description. In `schema/main.js`, modify the `queryType` object as follows:

```
const queryType = new GraphQLObjectType({
  name: 'RootQuery',
  fields: {
    hello: {
      type: GraphQLString,
      resolve: () => 'world'
    },
    diceRoll: {
      description: '**Simulate** a dice roll determined by count',
      type: new GraphQLList(GraphQLInt),
      args: {
        count: {
          type: GraphQLInt,
          defaultValue: 2
        }
      },
      resolve: (_, args) => {
        let rolls = [];
        for (let i = 0; i < args.count; i++) {
          rolls.push(roll());
        }
        return rolls;
      }
    }
  }
});
```

```
        }
        return rolls;
    },
},
usersCount: {
    description: 'Total number of users in the database',
    type: GraphQLInt,
    resolve: (_, args, { db }) =>
        db.collection('users').count()
    }
);
});
```

When we try to use the two fields that are now documented with a description property in GraphQL, that description appears in the dropdown:



Descriptions also show up in the **Docs** section in GraphiQL (upper right corner). This section is auto-generated from the GraphQL schema, and it shows a list of all fields defined on the RootQuery so far. For every field, we can see its type and description:

A screenshot of the GraphiQL Docs section for the "usersCount" field. The field is listed with its description "Total number of users in the database". Under the "TYPE" heading, the type "Int" is shown.

This reading of meta information about our schema is possible because of the introspective nature of GraphQL servers. We can use a GraphQL query to ask about the GraphQL schema and what capabilities that schema supports. For example, here's a query to ask about all fields of our example `RootQuery` object:

```
query TypeFields {
  __type(name: "RootQuery") {
    fields {
      name
      description
      args {
        name
      }
    }
  }
}
```

Here's the response we get for that:

```
{
  "data": {
    "__type": {
      "fields": [
        {
          "name": "hello",
          "description": null,
          "args": []
        },
        {
          "name": "diceRoll",
          "description": "**Simulate** a dice roll
                          determined by count",
          "args": [
            {
              "name": "count"
            }
          ]
        },
        {
          "name": "usersCount",
          "description": "Total number of users in the database",
          "args": []
        }
      ]
    }
  }
}
```

The `__type` is a built-in introspective field that should be available in any GraphQL implementation. The double underscores naming convention is reserved for the introspective system to avoid naming collisions with user-defined GraphQL types. Anything that starts with double underscores is part of the introspective API.

Notice how, in the server response for the `__type` query, the `hello` field has a `null` description because we did not define one there, and how only the `diceRoll` field accepts an argument. This is an example of the information a client tool, such as GraphiQL, can use to provide rich editing features to the user.



We can write Markdown in the description fields and GraphQL will use a Markdown renderer to display those descriptions. The GraphQL specification document encourages GraphQL tool writers to support Markdown rendering for the description field in their tools.

Another built-in field we can use to read more introspective information about the schema capabilities is the `__schema` field, which is available on the root type of a query. For example, if we didn't know the name of the `RootQuery` type, we can use the `__schema` field to find it:

```
query QueryTypeName {
  __schema {
    queryType {
      name
    }
  }
}
```

Here is the response for our example schema:

```
{
  "data": {
    "__schema": {
      "queryType": {
        "name": "RootQuery"
      }
    }
  }
}
```

We can also use the introspective API to read other capabilities of a GraphQL schema. For example, here's an introspective query to read the list of directives a GraphQL server supports:

```
query SchemaDirectives {
  __schema {
    directives {
      name
      description
      args {
        name
        description
      }
    }
  }
}
```

Since we have not implemented any custom directives for our example GraphQL schema, the server will respond with the built-in ones:

```
{
  "data": {
    "__schema": {
      "directives": [
        {
          "name": "include",
          "description": "Directs the executor to include
                          this field or fragment only
                          when the `if` argument is true.",
          "args": [
            {
              "name": "if",
              "description": "Included when true."
            }
          ]
        },
        {
          "name": "skip",
          "description": "Directs the executor to skip
                          this field or fragment
                          when the `if` argument is true.",
          "args": [
            {
              "name": "if",
              "description": "Skipped when true."
            }
          ]
        },
      ],
    }
  }
}
```

```
{  
  "name": "deprecated",  
  "description": "Marks an element of a GraphQL schema  
    as no longer supported.",  
  "args": [  
    {  
      "name": "reason",  
      "description": "Explains why this element  
        was deprecated, usually also  
        including a suggestion for how  
        to access supported similar data."  
    }  
  ]  
}  
}  
}  
}
```

Using the GraphQL introspection API, clients can read all the meta information about a GraphQL schema and use it to provide their users with rich features such as auto-complete and type warnings. The GraphQL JavaScript reference implementation library has a built-in `introspectionQuery` that we can use to ask for a complete representation of the server's type system. You can find this query if you search the library's GitHub repo (<https://github.com/graphql/graphql-js>) for `introspectionQuery`. To test the query, copy it into the GraphiQL instance for our example schema and you should see this big response:

```

1 query IntrospectionQuery {
2   __schema {
3     queryType { name }
4     mutationType { name }
5     subscriptionType { name }
6     types {
7       ...FullType
8     }
9     directives {
10       name
11       description
12       locations
13       args {
14         ...InputValue
15       }
16     }
17   }
18 }
19 fragment FullType on __Type {
20   kind
21   name
22   description
23   fields(includeDeprecated: true) { }
24   inputFields {
25     ...InputValue
26   }
27   interfaces {
28     ...TypeRef
29   }
30   enumValues(includeDeprecated: true) { }
31   possibleTypes {
32     ...TypeRef
33   }
34 }
35 fragment InputValue on __InputValue { }
36 fragment TypeRef on __Type { }

```

The right pane displays the JSON output of the introspection query:

```

{
  "data": {
    "__schema": {
      "queryType": {
        "name": "RootQuery"
      },
      "mutationType": null,
      "subscriptionType": null,
      "types": [
        {
          "kind": "OBJECT",
          "name": "RootQuery",
          "description": null,
          "fields": [
            {
              "name": "hello",
              "description": null,
              "args": [],
              "type": {
                "kind": "SCALAR",
                "name": "String",
                "ofType": null
              },
              "isDeprecated": false,
              "deprecationReason": null
            },
            {
              "name": "diceRoll",
              "description": "**Simulate** a dice roll determined by count",
              "args": [
                {
                  "name": "count",
                  "description": null,
                  "type": {
                    "kind": "SCALAR",
                    "name": "Int",
                    "ofType": null
                  },
                  "defaultValue": null
                }
              ],
              "type": {
                "kind": "LIST",
                "name": null,
                "ofType": {

```

*** #GitTag: chapter3-introspection ***

The type system

GraphQL is a strongly-typed language, and a GraphQL schema should have types for all objects that it uses. This includes properties on objects, arguments, and variables. The type system allows a GraphQL server to determine whether a query is valid at runtime.

Scalars and object types

The `GraphQLFieldConfig` object defines a field's type property to be `GraphQLObjectType`. An output type in GraphQL can be one of the following three things:

- A custom type, like the `EmployeeType` which we defined in a previous example.
- A `GraphQLScalarType`, which represents a scalar value that cannot have fields of its own. This could be one of the following:
 - `GraphQLInt` to represent an integer value
 - `GraphQLFloat` to represent a float value
 - `GraphQLString` to represent a string value
 - `GraphQLBoolean` to represent a Boolean value
 - `GraphQLID` to represent an identity value
- An instance of an object type class, like `GraphQLObjectType`. This is how we define nested relations in a GraphQL schema.

Interfaces and unions

Interfaces and unions are abstract types that can be used to group other types. We can use an Interface when there are common fields declared on the types of a group, and we can use a union when there are no common fields declared on the types of a group.

An Interface type in GraphQL defines the fields an implementation will contain, while a union type defines a list of different implementations. When we define a `GraphQLObjectType`, we can optionally list the interfaces that it implements using its `interfaces` property.

Let's say our schema defines an `EmployeeType` to represent the company's employees, and a `VendorType` to represent the company's vendors:

```
const EmployeeType = new GraphQLObjectType({  
  name: 'Employee',  
  fields: {  
    name: { type: GraphQLString },  
    departmentName: { type: GraphQLString },  
  }  
});  
  
const VendorType = new GraphQLObjectType({  
  name: 'Vendor',  
  fields: {  
    name: { type: GraphQLString },  
    vendorType: { type: GraphQLString },  
  }  
});
```

```
        name: { type: GraphQLString },
        companyName: { type: GraphQLString }
    }
});
```

Notice how both `EmployeeType` and `VendorType` define the field `name`. This is where a GraphQL interface can be useful. Let's name this interface `PersonType`. If `PersonType` defines the `name` field, we can say that both `EmployeeType` and `VendorType` implement this `PersonType` interface. Here's how we can define the `PersonType` interface:

```
const PersonType = new GraphQLInterfaceType({
    name: 'Person',
    fields: {
        name: { type: GraphQLString }
    }
});
```

With a `PersonType` interface defined, we can use the `interfaces` property on a `GraphQLObjectType` configuration object to provide a list of interfaces that an object implements:

```
const EmployeeType = new GraphQLObjectType({
    name: 'Employee',
    fields: {
        name: { type: GraphQLString },
        departmentName: { type: GraphQLString }
    },
    interfaces: [PersonType]
});

const VendorType = new GraphQLObjectType({
    name: 'Vendor',
    fields: {
        name: { type: GraphQLString },
        companyName: { type: GraphQLString }
    },
    interfaces: [PersonType]
});
```

The fact that an object implements an interface is our guarantee that the object will support all fields that are defined on that interface. When we ask this GraphQL schema about employees or vendors, we know that we can always ask about their name.

Let's add a new type that uses our `PersonType` directly:

```
const ContactType = new GraphQLObjectType({
  name: 'Contact',
  fields: {
    person: PersonType,
    phoneNumber: { type: GraphQLString },
    emailAddress: { type: GraphQLString }
  }
});
```

In a GraphQL query that asks about a contact object, we can ask about the person associated with that contact. Since we know that the interface defines a name, we can always ask about that:

```
query ContactQuery($contactId: Int!) {
  contact(contactId: $contactId) {
    person {
      name
    },
    phoneNumber,
    emailAddress
  }
}
```

Inside the `person` field, we can't directly ask for `departmentName` or `companyName` because a person could have one of those properties but not the other. However, we can use inline fragments to conditionally ask about them:

```
query ContactQuery($contactId: Int!) {
  contact(contactId: $contactId) {
    person {
      name
      ... on Employee {
        departmentName
      }
      ... on Vendor {
        companyName
      }
    },
    phoneNumber,
    emailAddress
  }
}
```

This query's response will include a `departmentName` but not a `companyName` when this contact is an employee, and it will include `companyName` but not `departmentName` when the contact is a vendor.

When we want to group two objects that don't have any fields in common with a certain logic, a GraphQL union is what we can use. For example, when we're modeling a resume object for employees, the resume will have different sections, and each section will have different fields, but the sections can be grouped under the category `Resume Section`.

Assume that we have the following two types in our schema:

```
const EducationType = new GraphQLObjectType({  
  name: 'Education',  
  fields: () => ({  
    schoolName: { type: GraphQLString },  
    fieldOfStudy: { type: GraphQLString },  
    graduationYear: { type: GraphQLInt }  
  })  
});  
  
const ExperienceType = new GraphQLObjectType({  
  name: 'Experience',  
  fields: () => ({  
    companyName: { type: GraphQLString },  
    title: { type: GraphQLString },  
    description: { type: GraphQLString }  
  })  
});
```

We can use a union to represent a resume section that can be either an education type or an experience type:

```
const ResumeSectionType = new GraphQLUnionType({  
  name: 'ResumeSection',  
  types: [ExperienceType, EducationType],  
  resolveType(value) {  
    if (value instanceof Experience) {  
      return ExperienceType;  
    }  
    if (value instanceof Education) {  
      return EducationType;  
    }  
  }  
});
```

When we have a union type in a GraphQL schema, we can use inline fragments to ask about the fields of the types that the union represents:

```
query ResumeInformation {
  ResumeSection {
    ... on Education {
      schoolName,
      fieldOfStudy
    }
    ... on Experience {
      companyName,
      title
    }
  }
}
```

Type modifiers

Any GraphQL type can be wrapped in one of these two type modifiers:

- **GraphQLList**: When we wrap other types with a `GraphQLList` instance, we are representing a list of those types. For example, to represent a list of integers, we defined our `diceRoll` field's type to be:

```
new GraphQLList(GraphQLInt)
```

If we want to represent a list of employees using the `EmployeeType` we defined in a previous example, we can do the following:

```
fields: {
  employees: {
    type: new GraphQLList(EmployeeType),
  }
}
```

- **GraphQLNonNull**: When we wrap other types with a `GraphQLNonNull` instance, we are representing the non-null version of those types. This wrapper enforces that the value it wraps is never null, and the type will raise an error if the value happens to be null. For example, an employee name should never be null, but the `boss` property can be null because at least one employee at a company does not have a boss:

```
const EmployeeType = new GraphQLObjectType({
  name: 'Employee',
  fields: () => ({
```

```
        name: { type: new GraphQLNonNull(GraphQLString) },
        boss: { type: EmployeeType },
    })
});
```

When we resolve the name field on an employee object, if a null value was found, GraphQL will raise an error.

These two type modifiers are also known as type makers because they make a new type, which wraps the original type. To read the original type in the introspective API, we can use the ofType property. For example, here's an introspection query to read all the types for a GraphQL schema:

```
query TypeFields {
  __schema {
    queryType {
      fields {
        name
        type {
          kind
          name,
          ofType {
            kind,
            name
          }
        }
      }
    }
  }
}
```

The response from our example GraphQL schema is as follows:

```
{
  "data": {
    "__schema": {
      "queryType": {
        "fields": [
          {
            "name": "hello",
            "type": {
              "kind": "SCALAR",
              "name": "String",
              "ofType": null
            }
          },
          {
            "name": "diceRoll",

```

```
        "type": {
          "kind": "LIST",
          "name": null,
          "ofType": {
            "kind": "SCALAR",
            "name": "Int"
          }
        }
      },
      {
        "name": "usersCount",
        "type": {
          "kind": "SCALAR",
          "name": "Int",
          "ofType": null
        }
      }
    ]
  }
}
```

Notice how both `hello` and `usersCount` have direct scalar types, while object of `diceRoll` type does not have a name. The object of `diceRoll` type is a list modifier. To see what type the object of `diceRoll` type modifies, we can read the `ofType` property, which only exists for the `diceRoll` field in this schema. This response tells us, using the type names, that `diceRoll` is a list of scalar integers.

Type modifiers can be combined. For example, we can use them to define a list of non-null items, a non-null list, or a list of lists.

Enums

When the scalar value that we want to represent for a field has a list of possible values in a set, and it can only be one of those values, we can represent the field in a GraphQL schema as an ENUM type.

For example, an employee's contract can be full-time, part-time, or shift-work. Let's assume we represented these values in our database with numbers; the database field `contract_type` has a check in `(1, 2, 3)`. Here's how we represent the contract type values in the GraphQL JavaScript implementation:

```
const ContractType = new GraphQLObjectType({  
  name: 'Contract',  
  values: {  
    FULLTIME: { value: 1 },  
    PARTTIME: { value: 2 },  
    SHIFTWORK: { value: 3 }  
  }  
});
```

ContractType is a new custom type that we can now use on the EmployeeType, for example like this:

```
const EmployeeType = new GraphQLObjectType({  
  name: 'Employee',  
  fields: {  
    name: { type: GraphQLString },  
    contractType: ContractType  
  }  
});
```

Let's introduce another type—a department type. For this type, we'll use the ContractType to represent the list of allowed contract types in a department:

```
const DepartmentType = new GraphQLObjectType({  
  name: 'Department',  
  fields: {  
    name: { type: GraphQLString },  
    contractTypes: new GraphQLList(ContractType),  
  }  
});
```

Every department will have a list of contract types that it can accept.

The resolve function

It's time to talk about the optional `resolve` function in a field configuration object:

```
type GraphQLFieldConfig = {  
  type: GraphQLOutputType;  
  args?: GraphQLFieldConfigArgumentMap;  
  resolve?: GraphQLFieldResolveFn;  
  deprecationReason?: string;  
  description?: ?string;  
}
```

We've already seen the `resolve` function in our example schema, where we used it to return data for each field. This function can accept four optional arguments:

```
type GraphQLFieldResolveFn = (
  source?: any,
  args?: {[argName: string]: any},
  context?: any,
  info?: GraphQLResolveInfo
) => any;
```

First argument – source

This argument represents the field we're configuring. For example, on the `EmployeeType`, we can define a `name` field to use the `source` argument value to read `firstName` and `lastName` from each employee object:

```
const EmployeeType = new GraphQLObjectType({
  name: 'Employee',
  fields: () => ({
    name: {
      type: GraphQLString,
      resolve: (obj) => `${obj.firstName} ${obj.lastName}`
    },
    boss: { type: EmployeeType },
  })
});
```

The `source` argument, which we named `obj` for this example, represents the employee object that we will be responding with. This `resolve` function does not need to use the other arguments. The `name` field here is an example of a computed field that does not need to be mapped to an object property.

Second argument – args

The value of this argument is an object that is associated with the `args` property that is defined on the field level (the same level where `resolve` is defined):

```
type GraphQLFieldConfig = {
  type: GraphQLOutputType;
  args?: GraphQLFieldConfigArgumentMap;
  resolve?: GraphQLFieldResolveFn;
  deprecationReason?: string;
  description?: ?string;
```

}

The `GraphQLFieldConfigArgumentMap` type is a simple object that holds a list of arguments:

```
type GraphQLFieldConfigArgumentMap = {
  [argName: string]: {
    type: GraphQLInputType;
    defaultValue?: any;
    description?: ?string;
  };
};
```

A field argument only requires a `type`, but we can also give it a `defaultValue` and a `description`.

We saw the arguments feature when we defined a `count` argument on the `diceRoll` field. Let's define some arguments for the `name` field on `EmployeeType`.

To test examples about `EmployeeType`, let's define an example employee object that we will use to resolve a top-level `exampleEmployee` field:

```
const exampleEmployee = {
  firstName: 'jane',
  lastName: 'doe'
};
```

In our `RootQuery` fields object in `schema/main.js`, we add a field for `exampleEmployee`:

```
exampleEmployee: {
  type: EmployeeType,
  resolve: () => exampleEmployee
},
```

We can now read our `exampleEmployee` data with a query like the following:

```
query EmployeeData {
  exampleEmployee {
    # Fields on EmployeeType
  }
}
```

Let's support an `upperCase` Boolean argument that would make the server resolve with the name in uppercase. Above the `queryType` object in `schema/main.js`:

```
const EmployeeType = new GraphQLObjectType({
  name: 'Employee',
  fields: () => ({
    name: {
      type: GraphQLString,
      args: {
        upperCase: { type: GraphQLBoolean }
      },
      resolve: (obj, args) => {
        let fullName = `${obj.firstName} ${obj.lastName}`;
        return args.upperCase ?
          fullName.toUpperCase() : fullName;
      }
    },
    boss: { type: EmployeeType }
  })
});
```

We first define the argument type on the `name` field—in our case it's a `GraphQLBoolean` (which we need to add to the `require('graphql')` constants). To use the user-supplied argument value, we read it from the second argument of the `resolve()` function itself. We can name this argument anything, but it's usually defined as `args` too.

Inside the `resolve()` function, `args.upperCase` will have the user-supplied value of the `upperCase` field argument. For example, here's a GraphQL query that uses aliases to read the name twice, once with normal format, and once with uppercase format:

```
query EmployeeNameCase {
  exampleEmployee {
    upperCaseName: name(upperCase: true)
    regularName: name(upperCase: false)
  }
}
```

The server response would be as follows:

```
{
  "data": {
    "exampleEmployee": {
      "upperCaseName": "JANE DOE",
      "regularName": "jane doe"
    }
  }
}
```

However, this makes the `name` field dependent on this `upperCase` argument; we can't query for `name` without supplying a value for `upperCase`. It would be nice if there was a default value. Also, what if we want to also support a lowercase version of the name, and keep the support for the regular title case? To solve both issues, let's introduce a new type for the `name` field. Instead of `upperCase` Boolean, we'll do a `letterCase` enum. In `schema/main.js`, define this new enum type:

```
const LetterCaseType = new GraphQLObjectType({  
  name: 'LetterCase',  
  values: {  
    TITLE: { value: 'title' },  
    UPPER: { value: 'upper' },  
    LOWER: { value: 'lower' }  
  }  
});
```

We'll need to add the `GraphQLObjectType` to the `require('graphql')` constants.

JavaScript does not have a `.toTitleCase()` function, so let's define a simple one. In `schema/main.js`:

```
const toTitleCase = str => {  
  return str.replace(/\w\S*/g, txt =>  
    txt.charAt(0).toUpperCase() + txt.substr(1).toLowerCase());  
};
```

To use the `LetterCaseType` enum, instead of replacing the current `name` field, let's add a new `nameForCase` field. Think of `nameForCase` as a new version for `name`. We can do something like the following:

```
const EmployeeType = new GraphQLObjectType({  
  name: 'Employee',  
  fields: () => ({  
    name: {  
      // ...  
    },  
    nameForCase: {  
      type: GraphQLString,  
      args: {  
        letterCase: { type: LetterCaseType }  
      },  
      resolve: (obj, args) => {  
        let fullName = `${obj.firstName} ${obj.lastName}`;  
        switch (args.letterCase) {  
          case 'lower':  
            return fullName.toLowerCase();  
        }  
      }  
    }  
  })  
});
```

```
        case 'upper':
            return fullName.toUpperCase();
        case 'title':
            return toTitleCase(fullName);
        default:
            return fullName;
    }
}
},
boss: { type: EmployeeType }
})
});
});
```

The `nameForCase` field can now be queried for in one of four ways:

```
query EmployeeNameCase {
  exampleEmployee {
    defaultCaseName: nameForCase
    titleCaseName: nameForCase(letterCase: TITLE)
    lowerCaseName: nameForCase(letterCase: LOWER)
    upperCaseName: nameForCase(letterCase: UPPER)
  }
}
```

Here's what the server will respond with:

```
{
  "data": {
    "exampleEmployee": {
      "defaultCaseName": "jane doe",
      "titleCaseName": "Jane Doe",
      "lowerCaseName": "jane doe",
      "upperCaseName": "JANE DOE"
    }
  }
}
```

Third argument – context

This argument represents a global context object that the GraphQL executor can pass to all resolver functions. It can be used, for example, to represent a database connection, an authenticated user session, or a reference to a request-specific cache object.

We've used the context object in Chapter 1, *An Introduction to GraphQL and Relay*, to pass a reference to the MongoDB connected `db` object to all resolver functions. Here's how we instructed the GraphQL executor to pass a context object to all resolver functions:

```
mongodb.MongoClient.connect(MONGO_URL, (err, db) => {
  ...
  app.use('/graphql', graphqlHTTP({
    schema: mySchema,
    context: { db },
    graphiql: true
  }));
  ...
});
```

In this example, the context object contains one property, the `db` object. We can add any other global context values as properties on this same context object.

We then used the context third argument in the `userCount` field resolve function:

```
usersCount: {
  description: 'Total number of users in the database',
  type: GraphQLInt,
  resolve: (_, args, { db }) => db.collection('users').count()
}
```

In this example, we destructured the `db` property out of the context object that the resolver function has access to.

Fourth argument – info

This argument represents a collection of information about the current execution state; we can use it, for example, to access the field name that we're currently resolving for, or the return type of the field we're currently resolving for, among a few other things.

The field name is helpful if we need to dynamically modify the resolved value of a field. For example, let's assume that we have an object in our schema with `first_name` and `last_name` properties, and we want to model our fields to be `camelCase` instead of `snake_case`; we can come up with a generic code that we can use for both `firstName` and `lastName` fields:

```
fields: {
  firstName: fromSnakeCase(GraphQLString),
  lastName: fromSnakeCase(GraphQLString),
}
```

```
// Assuming that we have a toSnakeCase() function
// Converts a camelCase string into snake_case
const fromSnakeCase = GraphQLType => {
  return {
    type: GraphQLType,
    resolve(obj, args, ctx, { fieldName }) {
      return obj[toSnakeCase(fieldName)];
    }
  };
};
```

In the `fromSnakeCase()` function, we use the resolver function's fourth argument, from which we destructured the `fieldName` value. Then, instead of returning the default resolver value, which is `obj[fieldName]`, we modify the resolver value to read the snake case property that matches the `fieldName` we're executing for:

```
*** #GitTag: chapter3-the-resolve-function ***
```

Resolving with promises

When we create APIs, we usually create them to work with some form of data, and when we work with data in Node.js, we do so asynchronously. We initiate a command to read or write data, and then we handle the response with a callback or a promise.

For example, to read the content of a file (the simplest form of a database) in Node.js, we do the following:

```
fs.readFile('/path/to/file', (err, data) => {
  if (err) throw err;
  console.log(data);
});
```

The callback function in the second argument will get executed asynchronously when the content of the file is ready. We can't use the return value of `fs.readFile` directly.

To work through a GraphQL example for working with promises, let's assume that we have a very simple database of inspirational quotes in a file in our project:

In `data/quotes`, we have the following:

```
The best preparation for tomorrow is doing your best today.  
Life is 10 percent what happens to you and 90 percent how you react to it.  
If opportunity doesn't knock, build a door.
```

We want to define a GraphQL field to return the most recent quote in our file (assuming that to be the last one).

The client would ask this query:

```
{ lastQuote }
```

The server should respond with the following:

```
{
  "data": {
    "lastQuote": "If opportunity doesn't knock, build a door."
  }
}
```

We can't simply resolve the GraphQL field with the result of `fs.readFile`. However, we can resolve it with a promise object that will resolve with the line of data that we want.

Let's first create a JavaScript promise that resolves with the last line of a given file, in `schema/main.js`:

```
const fs = require('fs');

const readLastLinePromise = path => {
  return new Promise((resolve, reject) => {
    fs.readFile(path, (err, data) => {
      if (err) throw reject(err);
      resolve(data.toString().trim().split('\n').slice(-1)[0]);
    });
  });
};
```

Here's how we would use this promise:

```
readLastLinePromise('data/quotes')
  .then(line => console.log(line))
  .catch(error => console.error(error));
```

To make a GraphQL `lastQuote` root field that will read the last quote from the file, we define a `lastQuote` field on the `RootQuery` object and resolve it with the promise itself. In `schema/main.js`:

```
const queryType = new GraphQLObjectType({
  name: 'RootQuery',
  fields: {
    lastQuote: {
      type: GraphQLString,
      resolve: () => readLastLinePromise('data/quotes')
```

```
    },
    // Other fields on RootQuery
}
};
```

The GraphQL executor is smart enough to see a promise returned and use its resolved value in the response for the query.

If we test the `{ lastQuote }` query, the server response will be as follows:

```
{
  "data": {
    "lastQuote": "If opportunity doesn't knock, build a door."
  }
}
```



We can also use promises to resolve a mutation operation. Let's make our GraphQL schema support an insert operation on our simple inspirational quotes file-based database.

We can construct a mutation request to insert a quote with a GraphQL string like this:

```
mutation {
  addQuote(body: "...")
}
```

GraphQL will execute the mutation, and it will return a JSON response. Usually, this JSON response will be related to the data that the mutation is mutating.

Let's create the mutation on the server. Similar to the query property, the mutation is just another `GraphQLObjectType`, and similar to the `lastQuote` query field, we need to respond with a promise that will resolve with an answer for the mutation. We can use `fs.appendFile` to append the new quote to our quotes file. In `schema/main.js`, add the following:

```
const appendLinePromise = (path, line) => {
  return new Promise((resolve, reject) => {
    fs.appendFile(path, line, err => {
      if (err) throw reject(err);
      resolve(line);
    });
  });
};

const mutationType = new GraphQLObjectType({
  name: 'RootMutation',
  fields: {
    addQuote: {
      type: GraphQLString,
      args: {
        body: { type: GraphQLString }
      },
      resolve: (_, args) =>
        appendLinePromise('data/quotes', args.body)
    }
  }
});
```

Then, modify the `mySchema` object to include this new `mutationType`:

```
const mySchema = new GraphQLSchema({
  query: queryType,
  mutation: mutationType
});
```

We gave the mutation property an identifying name `RootMutation`, and so far, we're only defining a single mutation, `addQuote`.

The `addQuote` mutation accepts a `body` field argument, which is a string. Then it passes the value of that argument to `appendLinePromise`.

`appendLinePromise` will return a promise that resolves with the same input after it successfully appends it to the quotes file.

Let's use this addQuote mutation capability now:

```
mutation {
  addQuote(body: "Try to be a rainbow in someone's cloud.")
}
```

The server response will be as follows:

```
{
  "data": {
    "addQuote": "Try to be a rainbow in someone's cloud."
  }
}
```



To verify that data was in fact persisted, we can follow the mutation operation with a read operation:

```
query {
  lastQuote
}
```

The server response will be as follows:

```
{
  "data": {
    "addQuote": "Try to be a rainbow in someone's cloud."
  }
}
```

A screenshot of the GraphiQL interface. On the left, the query is displayed:

```
1 query afterMutation {  
2   lastQuote  
3 }
```

On the right, the response is shown:

```
{  
  "data": {  
    "lastQuote": "Try to be a rainbow in someone's cloud."  
  }  
}
```

Below the query editor, there is a "QUERY VARIABLES" section.

*** #GitTag: chapter3-resolving-with-promises ***

Validation

The GraphQL executor will only execute requests that pass all validation rules. If there are any errors during the validation phase, a list of errors is returned instead of any response from executing the operations.

Validating a request is not just about the syntax; there are also rules that a GraphQL server should enforce even if the syntax of the request is correct. For example, we can't send a GraphQL server a document that has two queries with the same name, or a document that has one anonymous query and another named query.

There are also rules related to fields: we can't ask the server about properties that do not exist on objects, we can't use an alias name that matches another property on the field that we're asking for, and we can't have selection sets within scalar fields.

One of the most noticeable validation actions that you'll probably run into is when invalid types are used in the request document, for example, if a field expects a numeric argument and we send it a string instead.

The GraphQL executor will return clear errors when any violation of syntax or logic is sent in a request. The clarity of error messages is one of the great features of GraphQL.

Here's an example request for our example schema that would cause the GraphQL executor to respond with an error:

```
query validationExampleQuery {  
  diceRoll(count: "7")  
}
```

Since our `diceRoll` field expects a numeric count, supplying a string will cause the executor to halt the operation and return an error message instead:

```
{  
  "errors": [  
    {  
      "message": "Argument \"count\" has invalid value  
                  \"7\".\nExpected type \"Int\", found \"7\".",  
      "locations": [  
        {  
          "line": 2,  
          "column": 19  
        }  
      ]  
    }  
  ]  
}
```

The locations part of the error tells us exactly where in the query string this problem was encountered.

There are a lot of other validation rules in GraphQL. In fact, every feature in the language has some associated rules for validity. Besides the examples mentioned here, there are rules on using fragments, input values, directives, and variables.

Versioning

GraphQL has a unique perspective on versioning: *it can be avoided.*

Versioning complicates API usage and leaves the API designers with lots of decisions that need to be made. For example, should the new API be hosted on the same level of endpoints? What do we do about the old API endpoints? What if a client needs to partially use the new API while maintaining the old usage? How do we warn clients about deprecated parts of the API?

GraphQL's perspective is simple. Avoid versioning altogether. When you have new features that you need to push to new clients, just use new fields for them and keep the old fields as they are; everyone will be happy.

We've already followed this concept when we introduced the `nameFor` field in the examples about the resolver function arguments. We had a `name` field that expected a Boolean argument (`upperCase`), and we decided to support an enum argument instead (`letterCase`). Instead of replacing the field that we had already, we introduced a new field, `nameFor`.

If we want to stop supporting old fields in a schema, GraphQL has a feature to allow for deprecating those fields first. This gives the API users some time to learn about the deprecation and update their queries. We can deprecate a GraphQL field by adding a `deprecationReason` property on it. Let's deprecate the `name` field that we want to eventually stop supporting:

```
const EmployeeType = new GraphQLObjectType({
  name: 'Employee',
  fields: () => ({
    name: {
      type: GraphQLString,
      deprecationReason: 'Use nameFor instead',
      args: {
        upperCase: { type: GraphQLBoolean }
      },
      resolve: (obj, args) => {
        let fullName = `${obj.firstName} ${obj.lastName}`;
        return args.upperCase ?
          fullName.toUpperCase() : fullName;
      }
    },
    nameForCase: {
      // ..
    }
  })
});
```

Deprecated fields will continue to work as normal, but the tools that work with introspective queries will know about these deprecated fields and will possibly provide warnings about using them.

*** #GitTag: chapter3-versioning ***

Summary

To work with GraphQL on the server side, we need to write a valid schema that exposes the capabilities of our GraphQL server. We explored the schema object in this chapter and saw examples of how to define its elements.

We talked about the introspective nature of a GraphQL server and how we can use the GraphQL query language to read meta information about our schema.

We explored the GraphQL type system and the various types available for us to use to represent our data. We talked about the resolve function that we can use to map any user input to any logic we want, and we saw how the GraphQL executor accepts the use of promise objects in the resolve functions.

We talked about the various rules of validation that a GraphQL executer uses on all request documents, and we covered a bit of versioning strategies with GraphQL.

4

Configuring React Applications to Use Relay

In this chapter, we'll start building a React application that communicates with a GraphQL service using Relay. The first thing we need to do is to configure a Relay development environment for our React application. We'll use Webpack and Babel, set up a simple MongoDB-based GraphQL schema, read it directly with React, and then prepare our React application to work with Relay. The topics we'll cover in this chapter are:

- Working with MongoDB collections in GraphQL
- Setting up a Webpack environment with a Babel.js loader
- How to use a GraphQL endpoint directly with React applications
- Relay component containers
- Relay routes and root container

The example GraphQL schema

We'll start this example with the GraphQL schema we have been testing so far. Since we will only be talking to our MongoDB from this point forward, we can clean up all the example fields. I'll keep the `usersCount` field for a reference example to read from the database.

Here's our example GraphQL schema with only the `usersCount` root field:

```
const {  
  GraphQLSchema,  
  GraphQLObjectType,  
  GraphQLString,
```

```
GraphQLInt,
GraphQLList,
GraphQLBoolean,
GraphQLEnumType
} = require('graphql');

const queryType = new GraphQLObjectType({
  name: 'RootQuery',
  fields: {
    usersCount: {
      description: 'Total number of users in the database',
      type: GraphQLInt,
      resolve: (_ , args, { db }) => db.collection('users').count()
    }
  }
});

const mySchema = new GraphQLSchema({
  query: queryType
});

module.exports = mySchema;
```

Here's the `index.js` file that starts a simple Express.js server and exposes our schema in an HTTP interface:

```
const { MongoClient } = require('mongodb');
const assert = require('assert');
const graphqlHTTP = require('express-graphql');
const express = require('express');

const app = express();
const mySchema = require('./schema/main');
const MONGO_URL = 'mongodb://localhost:27017/test';

MongoClient.connect(MONGO_URL, (err, db) => {
  assert.equal(null, err);
  console.log('Connected to MongoDB server');

  app.use('/graphql', graphqlHTTP({
    schema: mySchema,
    context: { db },
    graphiql: true
  }));
}

app.listen(3000, () =>
  console.log('Running Express.js on port 3000')
);
```

```
});
```

This schema assumes the existence of a MongoDB collection named `users`, which is defined under a local database named `test`.

You can also check out branch `graphql-mongodb-start` from the book's GitHub repository (<https://github.com/edgecoders/learning-graphql-and-relay>) and you should get the simplified schema we have, without all the examples:

```
~/graphql-project $ git checkout graphql-mongodb-start
```

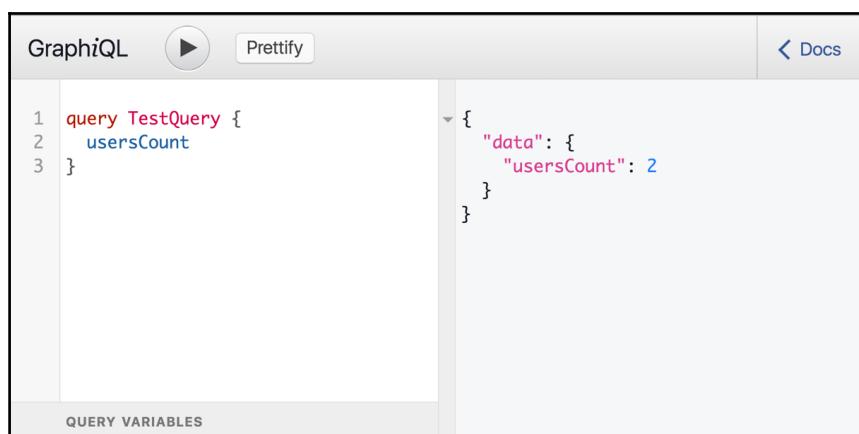
Run the `npm install` command, then make sure `mongod` is running and start the server with the `node index.js` command.

If everything works correctly, when you start the server, you should see the following:

```
~/graphql-project $ node index.js

Connected to MongoDB server
Running Express.js on port 3000
```

When you navigate to `http://localhost:3000/graphql` and read the `usersCount` field, you should see this answer:



The screenshot shows a GraphiQL interface. On the left, there is a code editor window containing the following GraphQL query:

```
1 query TestQuery {
2   usersCount
3 }
```

On the right, the results pane displays the JSON response:

```
{ "data": { "usersCount": 2 } }
```

```
*** #GitTag: chapter4-the-example-graphql-schema ***
```

The quotes library

Let's build a simple quotes library application that displays a list of short inspirational quotes and allows users to add more quotes to that list. This time, however, instead of managing the list of quotes in a file, we'll use a MongoDB collection.

In a mongo shell:

```
> db.createCollection("quotes")
{ "ok" : 1 }
```

We can use the `insertMany` collection function to insert our seed quotes:

```
> db.quotes.insertMany([
...   {
...     text: "The best preparation for tomorrow
...           is doing your best today",
...     author: "H. Jackson Brown"
...   },
...   {
...     text: "If opportunity doesn't knock, build a door",
...     author: "Milton Berle"
...   },
...   {
...     text: "Try to be a rainbow in someone's cloud",
...     author: "Maya Angelou"
...   },
... ])
```

The output of the previous command should be something like the following:

```
{
  "acknowledged" : true,
  "insertedIds" : [
    ObjectId("57281cc43a88dba20500f9c2"),
    ObjectId("57281cc43a88dba20500f9c3"),
    ObjectId("57281cc43a88dba20500f9c4")
  ]
}
```

Let's now create a simple GraphQL API for this collection. We can start with an `allQuotes` root field that returns all the quotes, but we first need to define a `QuoteType`. In the `schema/main.js` file, add the following:

```
const QuoteType = new GraphQLObjectType({
  name: 'Quote',
  fields: {
```

```
    id: {
      type: GraphQLString,
      resolve: obj => obj._id
    },
    text: { type: GraphQLString },
    author: { type: GraphQLString }
  }
});
```

Our `QuoteType` is a simple `GraphQLObjectType` that defines three `GraphQLString` fields. All fields on `QuoteType` are directly mapped to fields in the database, except for `id`, which we manually resolved with its object `_id` property. The `_id` property gets auto-generated by MongoDB when a new record is inserted. The other fields will be automatically resolved with their matching database object properties.

We can use this new `QuoteType` to define our planned `allQuotes` root field. Modify the `queryType` object in `schema/main.js`; we can get rid of the example `usersCount` field at this point:

```
const queryType = new GraphQLObjectType({
  name: 'RootQuery',
  fields: {
    allQuotes: {
      type: new GraphQLList(QuoteType),
      description: 'A list of the quotes in the database',
      resolve: (_, args, { db }) =>
        db.collection('quotes').find().toArray()
    }
  }
});
```

Since `allQuotes` will return a list of quotes, we need to modify the `QuoteType` and make a new type that is a `GraphQLList` with `QuoteType` items. For the `resolve` function, we use the MongoDB `find()` method on the `quotes` collection, which finds all the records in that collection. We then chain a `.toArray()` call to return a promise that will resolve to an array of quotes. GraphQL will work correctly with a promise-based answer, thanks to the smart execution engine behind it.

After a server restart, we can now use a GraphQL query to read our quotes from the database:



The screenshot shows a GraphiQL interface with a code editor on the left and a results panel on the right. The code editor contains the following GraphQL query:

```
1 {  
2   allQuotes {  
3     text,  
4     author  
5   }  
6 }  
  
{  
  "data": {  
    "allQuotes": [  
      {  
        "text": "The best preparation for tomorrow is doing your best today",  
        "author": "H. Jackson Brown"  
      },  
      {  
        "text": "If opportunity doesn't knock, build a door",  
        "author": "Milton Berle"  
      },  
      {  
        "text": "Try to be a rainbow in someone's cloud",  
        "author": "Maya Angelou"  
      }  
    ]  
  }  
}  
  
QUERY VARIABLES
```

*** #GitTag: chapter4-the-quotes-library ***

Setting up Webpack

We will be using Webpack with Babel.js plugins to work with our relay-based React applications. Webpack is a popular module bundler that supports a lot of extensions. We can use Webpack, for example, to work with Sass and CoffeeScript, and to prepare our JavaScript assets for production use. For the purpose of our example here, we'll only configure Webpack to enable us to write modern JavaScript (ES2015 and beyond), and to write in the JSX JavaScript extension.

Webpack is available as an npm package that we can install with the following:

```
~/graphql-project $ npm install -g webpack  
└── webpack@1.13.1
```

Once we have the `webpack` command globally available, we can use it to read a starting-point JavaScript file and bundle everything that file depends on using any extensions we want. Webpack will give us a single output file that we can write to any destination we choose.

Let's make our entry point `js/app.js` and have our `index.html` inside a `public` directory. We'll have Webpack bundle a single `bundle.js` file inside this `public` directory as well.

Here's a Webpack configuration file that will allow us to work with this structure. On the root level of the application, create a `webpack.config.js` file with this content:

```
const path = require('path');

module.exports = {
  entry: './js/app.js',
  output: {
    path: path.join(__dirname, 'public'),
    filename: 'bundle.js'
  },
  module: {
    loaders: [
      {
        test: /\.js$/,
        exclude: /node_modules/,
        loader: 'babel-loader'
      }
    ]
  }
};
```

We start with the entry point `js/app.js` and make Webpack generate a single file named `bundle.js` inside the `public` directory. While bundling, we're instructing Webpack to also use Babel on any file whose name ends with `.js`, excluding all files in the `node_modules` directory.

To make Babel work with ES2015, React, and all the proposed features for modern JavaScript, we can configure the following presets in a `.babelrc` file. On the root level of the application, create a `.babelrc` file with this content:

```
{
  "presets": [
    "react",
    "es2015",
    "stage-0"
  ]
}
```

To make all of this work, we need a few `npm` packages installed and saved locally:

```
~/graphql-project $ npm install --save webpack babel-loader \
    babel-preset-es2015 babel-preset-react babel-preset-stage-0
```

Let's create a simple React application to test our setup. We need an `index.html` file in the `public` directory:

```
<!DOCTYPE html>
<html>
<head>
  <title>Quotes</title>
  <link rel="stylesheet"
        href="https://maxcdn.bootstrapcdn.com/
        bootstrap/3.3.7/css/bootstrap.min.css" />
</head>
<body>
  <div id="react" class="container">
    Loading...
  </div>
  <script src="bundle.js"></script>
</body>
</html>
```

The only script we load here is `bundle.js`, which will be created by Webpack. We've also loaded the Bootstrap CSS framework in here. Bootstrap is optional, but it would make the quotes list look much better.

To make Express.js serve this `index.html` static file, we need to add the static middleware. Add this line to `index.js` right after we initialize the `app` constant:

```
app.use(express.static('public'));
```

Here's a simple example React application that uses a modern JavaScript feature (Class static properties). Create a new `app.js` file under a `js` directory with this content:

```
import React from 'react';
import ReactDOM from 'react-dom';

class App extends React.Component {
  static defaultProps = {
    greeting: 'Hello'
  };
  render() {
    return (
      <div>
        {this.props.greeting} World
      </div>
    );
  }
}

ReactDOM.render(<App />, document.getElementById('root'));
```

```
        </div>
    );
}
}

ReactDOM.render(
  <App />,
  document.getElementById('react')
);
```

Note how this code uses the new JavaScript module syntax import/export. The webpack process will be able to understand and work with this syntax through the Babel presets.

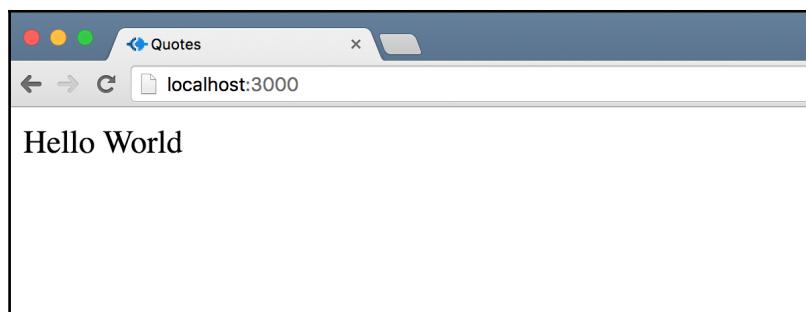
Since we're depending on `react` and `react-dom` packages here, we need to install them:

```
~/graphql-project $ npm install --save react react-dom
├── react@15.3.0
└── react-dom@15.3.0
```

Finally, to generate the `bundle.js` file, we need to invoke the `webpack` command:

```
~/graphql-project $ webpack
Hash: e911bf615a0446db8668
Version: webpack 1.13.1
Time: 992ms
      Asset      Size  Chunks             Chunk Names
bundle.js  727 kB       0  [emitted]  main
+ 172 hidden modules
```

After restarting the server, if everything works, when we navigate to `http://localhost:3000/` we should now see a **Hello World** message:



```
*** #GitTag: chapter4-setting-up-webpack ***
```

Using GraphQL without Relay in React applications

If we don't need any of the features that Relay provides for us, we can still use GraphQL directly in a React application using a simple Ajax library. For this example, we'll use the modern `fetch` library that browsers have started to support natively.

We first need to build our React components. Let's start with a `QuotesLibrary` component that lists an array of `Quote` components.

The `QuotesLibrary` component is a simple one; it will load the list of quotes when it mounts, and map them into an array of `Quote` components.

To keep things simple, we'll make the `QuotesLibrary` component our top-level one and define it in `js/app.js`:

```
import React from 'react';
import ReactDOM from 'react-dom';

import Quote from './quote';

class QuotesLibrary extends React.Component {
  state = { allQuotes: [] };

  componentDidMount() {
    // Load the quotes list into this.state.allQuotes
  }

  render() {
    return (
      <div className="quotes-list">
        {this.state.allQuotes.map(quote =>
          <Quote key={quote.id} quote={quote} />
        )}
      </div>
    )
  }
}

ReactDOM.render(
  <QuotesLibrary />,
  document.getElementById('react')
);
```

Assuming we can successfully load the list of quote objects into the `QuotesLibrary` component's state from within a `componentDidMount()` method, displaying them is a matter of mapping that array as an array of `Quote` elements.

Here's a simple implementation of the `Quote` component, which is also on the same level under `js/quote.js`, add the following:

```
import React from 'react';

class Quote extends React.Component {
  render() {
    return (
      <blockquote>
        <p>{this.props.quote.text}</p>
        <footer>{this.props.quote.author}</footer>
      </blockquote>
    );
  }
}

export default Quote;
```

Now all we need to do is figure out how to use a GraphQL query to make the list of quotes available to the `QuotesLibrary` component. Since `fetch` is available natively in modern browsers, let's just use that in the `componentDidMount()` method of the `QuotesLibrary` component.



For browsers that don't support `fetch` natively, we can use this polyfill by GitHub: <https://github.com/github/fetch>. This repository also has excellent instructions on how to use the `fetch` method. If you're using the latest Chrome, Firefox, Edge, or Opera, `fetch` should be available.

We can fetch our data with this simple GraphQL query:

```
// In the QuotesLibrary component in js/app.js
componentDidMount() {
  // Load the quotes list into this.state.allQuotes
  fetch(`graphql?query={\n    allQuotes {\n        id,\n        text,\n        author\n    }\n}`)\n    .then(response => response.json())\n    .then(json => this.setState(json.data))
```

```
.catch(ex => console.error(ex))  
}
```

Our GraphQL endpoint is responding on HTTP under `/graphql`, and we can use the `query` parameter to send it our GraphQL queries. The query we used here is a simple one that reads all the quotes' information, including their MongoDB-generated `id`. The `QuotesLibrary` component uses an `id` field on every quote for the React `key` property.

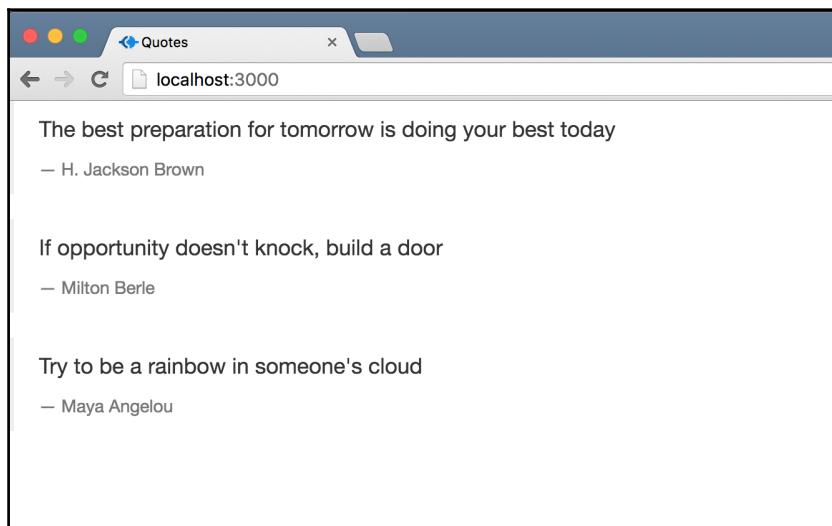
Once we have the JSON object returned from the server, we can use its `data` property to set the state of the `QuotesLibrary` component and update the `allQuotes` array.



Don't forget to `webpack` when you change any JavaScript files; I always run `webpack` with the `-w` (`--watch`) flag in development, which will take care of that.

You should also use the `-d` flag, which will generate and include source maps with the bundled file.

That's everything we need. Here's what you should see now when you go to `http://localhost:3000/`:



*** #GitTag: chapter4-using-graphql-without-relay ***

Relay containers

The first thing we need to do in order for our React application to use Relay is to wrap our components with the higher-order Relay container component that makes the React component understand Relay features. To be able to do that, we need to import the Relay library. In both `js/app.js` and `js/quote.js`, add the following:

```
import Relay from 'react-relay';
```

This import line will be needed in every file that defines a React component because we will convert every React component into a Relay-contained component. Add the same line to `js/quote.js`.

We'll need to install this new dependency for `webpack` to find it:

```
~/graphql-project $ npm install --save react-relay \
  babel-relay-plugin
  └── react-relay@0.9.2
  └── babel-relay-plugin@0.9.2
```



We will be using the `babel-relay-plugin` in the next chapter, the package `react-relay` depends on it.

Starting with our top-level `QuotesLibrary` component in `js/app.js`, here's how to wrap it with the Relay container higher-order component:

```
QuotesLibrary = Relay.createContainer(QuotesLibrary, {
  fragments: {}
});
```

This is a barebones Relay container; it generates a new React component that also understands Relay features. We'll learn how to work with the `fragments` property in the next chapter, but first, let's see how to render a Relay-contained React component into the DOM. We first need a `Relay.Route` object to define the entry points into the Relay application.

After defining the Relay container for `QuotesLibrary` in `js/app.js`, add this:

```
class AppRoute extends Relay.Route {
  static routeName = 'App';
}
```



The term `route` here might be a bit confusing. Relay routes have nothing to do with URL routing.

We can use this `AppRoute` along with the top-level `QuotesLibrary` component to render a `Relay.RootContainer` component.

Replace the previous `ReactDOM.render` call in `js/app.js` with the following:

```
ReactDOM.render(
  <Relay.RootContainer
    Component={QuotesLibrary}
    route={new AppRoute()}
  />,
  document.getElementById('react')
);
```

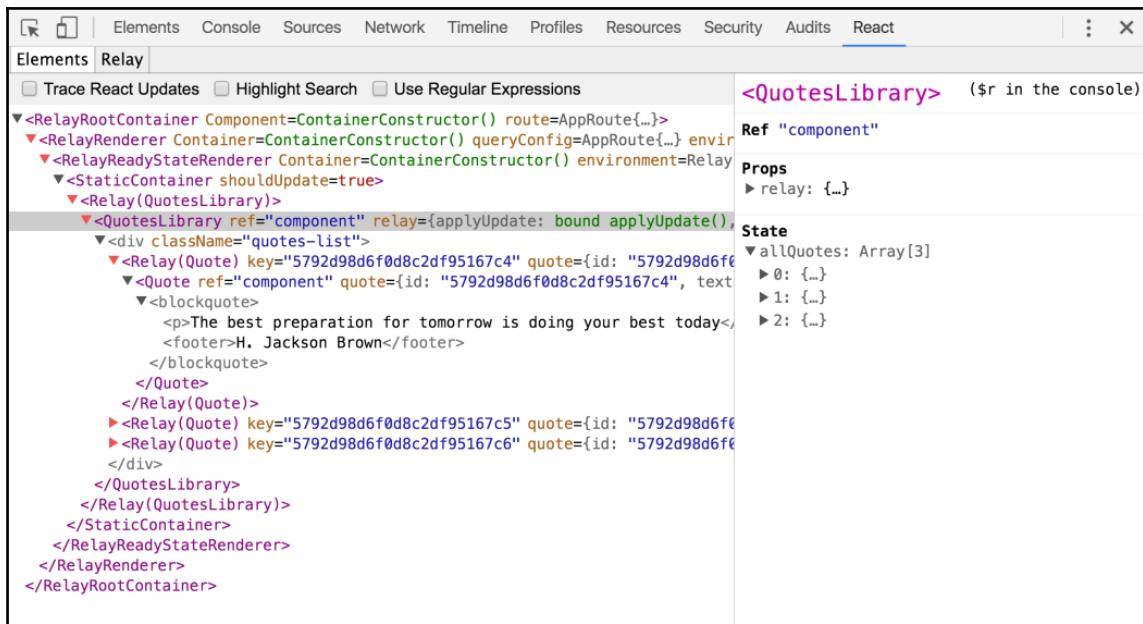
`Relay.RootContainer` also generates a React component, and it has some similarities to `ReactDOM.render` itself. It needs something to render (the `component` property), and where to start in the process (the `route` property). The component to render is our Relay-contained `QuotesLibrary` component, and the `route` object is where we can put instructions for Relay about where in the GraphQL graph it should start its querying. We will write these instructions in the next chapter, but now we can go ahead and test that our Relay-contained `QuotesLibrary` component will still render correctly with this setup.

We can do the same Relay container wrapping call for the `Quote` component:

```
// in js/quote.js
Quote = Relay.createContainer(Quote, {
  fragments: {}
});
```

This will not affect the rendering of our app so far. Relay-contained React components work just like normal React components, so we can use them the same way. Webpack and restart the server to test.

Although the rendered content at this point looks exactly the same, we're actually rendering a lot more components than we did before. Take a look at the React tab in your developer tools (assuming you have the React devtools extension; if you don't, you should: <https://github.com/facebook/react-devtools>):



Notice how we have a top-level `RelayRootContainer` now, which renders a `RelayRenderer`, which renders a `RelayReadyStateRenderer`, which renders a `StaticContainer` component, which in turns renders our React application. All of our React components are wrapped in a `Relay (Component)` React component. Our application is now ready to express its data requirements with Relay.

*** #GitTag: chapter4-relay-containers ***

Summary

In this chapter, we started building a React application. It's a simple application that so far renders a list of inspirational quotes using a direct GraphQL query with an AJAX request. We've also prepared our React application to use Relay by converting all React components into Relay-contained components. We've seen how to use Relay's root container to render a Relay-contained component in the DOM. We're about ready to make our GraphQL query through Relay itself, which is exactly what we will be doing in the next chapter.

5

Making GraphQL Queries Relay-Compliant

In this chapter, we'll continue building our inspirational quotes Relay application, which we started in the previous chapter. Before we can use our GraphQL schema with Relay directly, there are a few things we need to do first to make it Relay-compliant. For example, we should match our GraphQL hierarchy to our React components hierarchy, and to do any paginations on the data, we should implement Relay's connection model.

The topics we'll cover in this chapter are:

- Transforming GraphQL queries for Relay
- Matching React's component hierarchy in GraphQL
- First query operation with Relay
- Edges and nodes in Relay's connection model

Transforming GraphQL queries for Relay

Relay is a JavaScript framework that's inspired by React itself. With React, instead of working with HTML strings, we work with JavaScript objects. This gives React much more power and flexibility, and makes the code more readable. Relay has a similar approach to working with GraphQL queries: we need to first convert them into JavaScript objects.

When we have an object that represents a GraphQL query, we can add more information about the query to that object. For example, we can use the introspective API to analyze the query, validate the field types, and add those types to the object for Relay.

For our simple AllQuotes query:

```
query AllQuotes {  
  allQuotes {  
    id  
    text  
    author  
  }  
}
```

Here's how Relay wants to represent it as an object (this is a simplified version):

```
{  
  "name": "AllQuotes",  
  "fieldName": "allQuotes",  
  "type": "Quote",  
  "children": [  
    {  
      "fieldName": "id",  
      "type": "String"  
    },  
    {  
      "fieldName": "text",  
      "type": "String"  
    },  
    {  
      "fieldName": "author",  
      "type": "String"  
    }  
  ]  
}
```

This object represents the same information as the query string, but with more details. For example, using the object, we know that the three scalar fields on a single quote are all strings.

Luckily, we don't have to do this object conversion manually for GraphQL operations; Relay has a helper for that. It's called `Relay.QL`, and it's a tag that we can use to make our query string a *Tagged Template Literal*.

In JavaScript, a template tag is a function, whose name (the tag) can be used to prefix any template string, and it acts just like a normal function with a little bit more awareness about the structure of its single input (which is the string template). When we define the tag function, the arguments for that function contain an array of strings and an array of values.

In a node REPL (entering the `node` command by itself), try the following code snippet:

```
const INT = (strings, ...variables) => {
  console.log(`Ignoring ${variables}`);
  return strings.join('interesting');
}
const adj = 'new';
console.log(INT `JavaScript has ${adj} features`);
```

When Node executes this code, instead of logging *JavaScript has new features*, here's what we will see:

```
Ignoring new
JavaScript has interesting features
```

This happened because the `INT` template tag intercepted the template literal and modified the output by dropping the actual variables values and using custom logic on the template string instead.

Relay uses this feature in JavaScript to intercept a template that represents a GraphQL string and return an object instead of a string:

```
Relay.QL `query { ... }` => {}
```

Since this helper adds type information to the object representing the query, it needs access to the GraphQL schema. Instead of caching the whole schema on the client (which might be a bad idea for a big schema), Relay uses a Babel plugin on the server to prepare these `Relay.QL` objects during the compile time of all JavaScript resources.

For our `AllQuotes` query, here's what we need to do:

1. First, we use the `Relay.QL` `` template syntax to write all GraphQL queries for clients.
2. Then, we process all the JavaScript resources (with webpack).
3. When we have a `Relay.QL` call for our `AllQuotes` query, we ask the GraphQL schema about the types for the `allQuotes`, `id`, `text`, and `author` fields.
4. Then, we use these types to generate the desired query object and return it.

While this sounds simple, it does require a little bit of configuration to make it efficient.

A Relay application will have multiple `Relay.QL` calls. Instead of asking the GraphQL server about field types every time we want to convert a query, we can optimize this server process by caching the full schema structure into a big JSON object on the server and using the cache for the `Relay.QL` calls.

This will speed up the webpack bundle process, but it comes with the disadvantage that we need to update the cache when the server schema changes.

Since we read the schema in our `index.js` file (to make it available through an HTTP interface), let's make the server generate this cached JSON schema for us in the same file. This way, when we change the schema, we'll just need to restart the server to update the cache.



The schema JSON cache generation task can be a separate one from the task to start the server. If we do that, we can refresh the cache independently from the server. We're putting these two tasks in the same place for simplicity. In *Chapter 8, Deploying to the Cloud*, we will be splitting them into different files.

We'll write a function that prepares the cache by reading a full introspective query about our schema. The GraphQL JavaScript implementation has an `introspectiveQuery` utility variable that we can use for that. Once we have the output of this full introspective query (which is the JSON cache we are after), we'll just write it to a file on the system.

We can use the `fs` node library along with the `path` library to write the cache file to the filesystem. Let's first import them. In `index.js` add the following:

```
const fs = require('fs');
const path = require('path');
```

We also need to import the `introspectiveQuery` from the `graphql` npm package. It's available under `utilities`:

```
const { introspectionQuery } = require('graphql/utilities');
```

If you're curious, add a `console.log(introspectionQuery)` line after the `require` line to see `introspectiveQuery` full value; you can actually copy that output and run it in GraphiQL to see the JSON cache we are about to generate:

The screenshot shows the GraphiQL interface with an introspection query on the left and its resulting JSON response on the right.

```

1 query IntrospectionQuery {
2   __schema {
3     queryType {
4       name
5     }
6     mutationType {
7       name
8     }
9     subscriptionType {
10    name
11   }
12   types {
13     ...FullType
14   }
15   directives {
16     name
17     description
18     locations
19     args {
20       ...InputValue
21     }
22   }
23 }
24
25 fragment FullType on __Type { }
26 fragment InputValue on __InputValue { }
27 fragment TypeRef on __Type { }
100

```

QUERY VARIABLES

The JSON response on the right is a complex object representing the schema. It includes fields for the schema itself, types (with their names, descriptions, and fields), and specific fields like 'allQuotes' which are described as a list of quotes in the database.

We can use the same `graphql()` function – which we have previously used to create the readline interface for the GraphQL schema – to ask our schema for its response for the `introspectiveQuery`.

Import this `graphql()` function again (we removed it when we changed the interface to use `graphQLHTTP`). In `index.js` add the following:

```
const { graphql } = require('graphql');
```

Then, place this snippet of code right before we make the express app listen on port 3000:

```

graphql(mySchema, introspectionQuery)
  .then(result => {
    fs.writeFileSync(
      path.join(__dirname, 'cache/schema.json'),
      JSON.stringify(result, null, 2)
    );
    console.log('Generated cached schema.json file');
  })
  .catch(console.error);

```

The `graphql()` call returns a promise. If resolved correctly, the promise will give us access to a `result` JSON object. To write the `result` object to a file, we can use the `writeFileSync` function from the `fs` library, convert the content of the `result` object into a more readable string using `JSON.stringify` with two-space indentation for readability, and use the stringified version as the content of our cache file.

We also need to create this new `cache` directory before we proceed:

```
~/graphql-project $ mkdir cache
```

When we start the node server now, we should see the new `console.log` message:

```
~/graphql-project $ node index.js
Connected to MongoDB server
Running Express.js on port 3000
Generated cached schema.json file
```

Under the `cache` directory, we should see the newly generated `schema.json` file.

Here are the first 20 lines of this new big file:

```
~/graphql-project $ head -20 cache/schema.json
{
  "data": {
    "__schema": {
      "queryType": {
        "name": "RootQuery"
      },
      "mutationType": null,
      "subscriptionType": null,
      "types": [
        {
          "kind": "OBJECT",
          "name": "RootQuery",
          "description": null,
          "fields": [
            {
              "name": "allQuotes",
              "description": "A list of the quotes
in the database",
              "args": [],
              "type": {
                "kind": "LIST",
                "of": "String"
              }
            }
          ]
        }
      ]
    }
  }
}
```



Don't forget to add `cache/schema.json` to the source control ignore list. This is a generated big file that will just add noise to your source control.

With our schema cached as a JSON file, when we call the `webpack` command now, we can read the cached content once and re-use it from memory to speed up the calls to `Relay.QL`. We can do this with a Babel plugin.

Babel plugins provide an easy way for us to invoke custom behavior in the `webpack/babel-loader` process that bundles all our JavaScript files into a single `bundle.js` file.

The plugin we need is published as the `babel-relay-plugin` npm package; we've already installed this package when we installed Relay.

Here's how we can prepare this plugin for Babel using our own schema. Create a `babelRelayPlugin.js` file on the root level and put the following code in it:

```
const babelRelayPlugin = require('babel-relay-plugin');
const schema = require('./cache/schema.json');

module.exports = babelRelayPlugin(schema.data);
```

These lines will simply import the babel relay plugin and use it against our schema JSON cached data. It then exports a function that is ready to be plugged into the Babel process. To make use of this new plugin, we can add it to the `.babelrc` file. Replace the content of the current `.babelrc` file with the following:

```
{
  "passPerPreset": true,
  "presets": [
    {"plugins": ["./babelRelayPlugin"]},
    "react",
    "es2015",
    "stage-0"
  ]
}
```

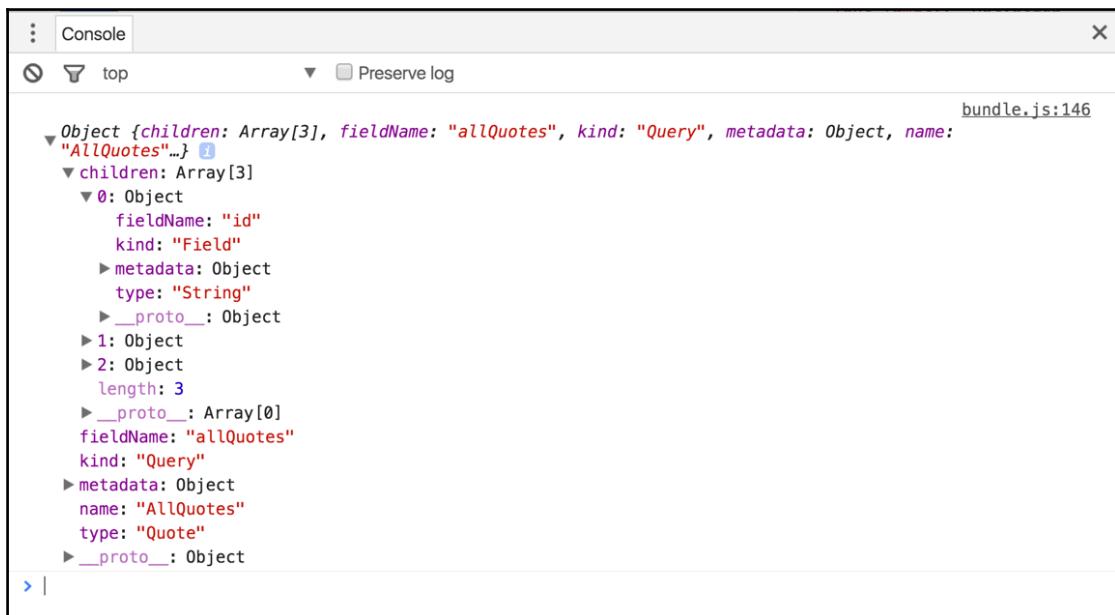


The `passPerPreset` option tells Babel to execute the presets one after the other; this way, we make sure that the Relay plugin is executed first, then the other presets get executed in order after that. However, this is an experimental feature in Babel and its syntax and purpose might change.

With this configuration, when we run the webpack command now, it will convert all Relay.QL tagged queries into objects. To test that this operation actually works now, we can put this `console.log` line anywhere in `js/app.js`:

```
console.log(
  Relay.QL `query AllQuotes {
    allQuotes {
      id
      text
      author
    }
  }``,
);
```

Run `webpack` after adding that line, restart the Node server, then refresh the browser. Here's the output we should see:



```
*** #GitTag: chapter5-transforming-graphql-queries-for-relay ***
```

Root-level field for the quotes library

Our React application component hierarchy is `QuotesLibrary` → `Quote`. Both `QuotesLibrary` and `Quote` represent a component that has data requirements. In the previous chapter, when we fetched our GraphQL query with an AJAX fetch operation, we specified all the requirements we needed in the same place; however, different components in a React application can tell us different parts of the requirements they care about. We can come up with these partial requirements by simply looking at the variables every component uses in its `render()` function:

```
// Quote's render function
render() {
  return (
    <blockquote>
      <p>{this.props.quote.text}</p>
      <footer>{this.props.quote.author}</footer>
    </blockquote>
  );
}
```

The `Quote` component requires a `quote` data object (because it uses `this.props.quote`). It also requires that this object defines both the `text` field and the `author` field (because it uses these two properties on its `quote` object):

```
// QuoteLibrary's render function
render() {
  return (
    <div className="quotes-list">
      {this.state.allQuotes.map(quote =>
        <Quote key={quote.id} quote={quote} />)
    </div>
  );
}
```

The `QuotesLibrary` component requires a collection of `quote` objects (because it uses the `this.state.allQuotes` array). It also requires that every object in this collection defines an `id` field (because it uses `quote.id` inside the `map()` call).

The `QuotesLibrary` component does not really care about all the other properties of a `quote` object. These properties are simply not the `QuotesLibrary` component's responsibility. Similarly, a `Quote` component does not care whether we're fetching a list of quotes or a just a single random quote. It also does not care about the `id` property for a `quote` (because `id` is not used in `Quote`). All the `Quote` component cares about is that we have a single `quote` object and that the object has both the `text` and the `author` properties.

This separation of data requirement ownership is helpful. It allows the components to grow independently, allows us to debug the code more easily, and makes testing in general easier.

We can use GraphQL fragments to express the data requirements for every component. For example, the fragment that the `Quote` component can use to express its data requirement would be:

```
fragment OneQuote on Quote {  
  text  
  author  
}
```

A fragment is defined on a GraphQL type, such as `Quote`. However, we have not defined a type for our quotes library itself. So far, we've been reading the `allQuotes` fields from the root level directly:

The screenshot shows the GraphiQL interface with a query editor and a results viewer. The query is:

```
1 {  
2   allQuotes {  
3     text,  
4     author  
5   }  
6 }
```

The results pane displays the JSON response:

```
{  
  "data": {  
    "allQuotes": [  
      {  
        "text": "The best preparation for tomorrow is doing your best today",  
        "author": "H. Jackson Brown"  
      },  
      {  
        "text": "If opportunity doesn't knock, build a door",  
        "author": "Milton Berle"  
      },  
      {  
        "text": "Try to be a rainbow in someone's cloud",  
        "author": "Maya Angelou"  
      }  
    ]  
  }  
}
```

Below the results, there is a section labeled "QUERY VARIABLES".

The Relay-recommended structure is to associate every React component with its own object, which is represented with a fragment on a custom type in the GraphQL API.

To represent our `QuotesLibrary` component, let's create a `QuotesLibrary` GraphQL type in our `schema/main.js`:

```
const QuotesLibraryType = new GraphQLObjectType({  
  name: 'QuotesLibrary',  
  fields: {
```

```
    allQuotes: {
      type: new GraphQLList(QuoteType),
      description: 'A list of the quotes in the database',
      resolve: (_, args, { db }) =>
        db.collection('quotes').find().toArray()
    }
  );
});
```



It's a good idea to match the type names to the React components that are using them, but it is not required.

You can imagine, later in the lifetime of this project, we might introduce other fields on this `QuotesLibrary` type. For example, we might have a `randomQuote` field, or a `top10Quotes` field.

With the `QuotesLibraryType` defined, we can now change our root-level GraphQL field to respond with an object of type `QuotesLibrary`. Let's name it `quotesLibrary`. We need to resolve this root-level field with something, and since we are resolving the `allQuotes` property directly from the `QuotesLibraryType` itself, we can simply resolve the root-level field `quotesLibrary` with an empty object. Create an empty object and replace the content of `queryType`:

```
const quotesLibrary = {};  
  
const queryType = new GraphQLObjectType({  
  name: 'RootQuery',  
  fields: {  
    quotesLibrary: {  
      type: QuotesLibraryType,  
      description: 'The Quotes Library',  
      resolve: () => quotesLibrary  
    }  
  }  
});
```



There are many other options for structuring our root-level and children resolving objects. We can, for example, define a class with methods that return the objects to be resolved. An empty object on the root level is picked here for simplicity.

Now, to read the `allQuotes` field, we'll have to start on the root level with a `quotesLibrary` field. Restart the server and test the new GraphQL structure with the following:



The screenshot shows a GraphiQL interface with a query editor and a results viewer. The query is:`1 {
2 quotesLibrary {
3 allQuotes {
4 text
5 author
6 }
7 }
8 }`

The results pane shows the JSON response:`{
 "data": {
 "quotesLibrary": {
 "allQuotes": [
 {
 "text": "The best preparation for tomorrow is doing your best today",
 "author": "H. Jackson Brown"
 },
 {
 "text": "If opportunity doesn't knock, build a door",
 "author": "Milton Berle"
 },
 {
 "text": "Try to be a rainbow in someone's cloud",
 "author": "Maya Angelou"
 }
]
 }
 }
}`

Below the results pane, there is a section labeled "QUERY VARIABLES".

We now have a GraphQL type on every level to represent the two component levels in our app. The `QuotesLibraryType` to represent the `QuotesLibrary` component and the `QuoteType` to represent the `Quote` component.

The GraphQL fragment we can now use to represent our `QuotesLibrary` component's data requirements would be:

```
fragment AllQuotes on QuotesLibrary {  
  allQuotes {  
    id  
    ...OneQuote  
  }  
}
```

Notice how this `AllQuotes` fragment now exactly mirrors the data requirements for the `QuotesLibrary` React component, including the fact that there are sub-requirements (for every `Quote` component) to be decided by the `Quote` component (using the `OneQuote` fragment).

That's the basic idea of the colocation concept in Relay; every component will give Relay a fragment for its data requirements. Those fragments can include each other, like how the `AllQuotes` fragment included the `OneQuote` component. Relay will put these fragments together and form a single GraphQL operation for the application starting point.

```
*** #GitTag: chapter5-root-level-field-for-the-quotes-library ***
```

First query operation with Relay

Our GraphQL fragments are ready to be hooked into Relay! All we need to do now is define them in the new Relay containers with which we wrapped our React components in the previous chapter.

Starting with the `Quote` component, we can tell Relay about its data requirements using the `fragments` property in the `createContainer` second argument.

Modify the `Relay.createContainer` call in `js/quote.js` to the following:

```
Quote = Relay.createContainer(Quote, {
  fragments: {
    quote: () => Relay.QL`fragment OneQuote on Quote {
      text
      author
    }`,
  }
});
```

This `fragments` property expects an object with one or more fragments; every fragment is represented with a name, which is a key on that object, and a value that is a function which returns a `Relay.QL` tagged fragment. Our `OneQuote` fragment is what we tag and return for the `quote` property on the `fragments` property.

A component can specify multiple named fragments; when the data represented by a named fragment become available, Relay will make it available to that component using the `component props` object. The data represented by the `quote` named fragment will be available to the `Quote` component using `this.props.quote`.

Similarly, we can tell Relay about the data requirements for the `QuotesLibrary` component using the `fragments` property on the container we used to wrap the `QuotesLibrary` component.

Modify the `Relay.createContainer` call in `js/app.js` to the following:

```
QuotesLibrary = Relay.createContainer(QuotesLibrary, {
  fragments: {
    library: () => Relay.QL`fragment AllQuotes on QuotesLibrary {
      allQuotes {
        id
        ${Quote.getFragment('quote')}
      }
    }
  }
});
```

Notice the new syntax to fetch a sub-fragment:

```
  ${Quote.getFragment('quote')}
```

This means, in that exact place inside the `AllQuotes` fragment, we're going to ask the `Quote` React component about its requirements for a `quote` object. The `Quote` component is the one that determines the fields for every `quote` object we're interested in here (`text` and `author` in this example). The `QuotesLibrary` component does not even need to know this information. In fact, Relay will not even make this information available to the `QuotesLibrary` component at all; the `text` and `author` properties on every `quote` will only be available to the `Quote` component itself because it was the one that asked for them.

When the data requested by the `QuotesLibrary` component becomes available globally in the app, Relay will make it accessible to `QuotesLibrary` using its `props`. The `library` fragment will be available as `this.props.library`, and we know that this `library` object will have an `allQuotes` property that holds our array of quotes.

We'll need to modify the way we read the array of quotes inside the `render()` function for `QuotesLibrary` to match this new hierarchy of properties.

Modify the class definition for `QuotesLibrary` in `js/app.js` to the following:

```
class QuotesLibrary extends React.Component {
  render() {
    return (
      <div className="quotes-list">
        {this.props.library.allQuotes.map(quote =>
          <Quote key={quote.id} quote={quote} />
        )
      </div>
    )
  }
}
```

The state object that we previously had on this component is not needed anymore. Relay will manage the application state for us.

The AJAX fetch code that we previously wrote in `componentDidMount()` is also not needed anymore. Relay will do all the AJAX fetching for us.

Finally, we need to make our `AppRoute` class construct the global GraphQL query that Relay can send to the server. We can't send those fragments by themselves to the GraphQL server; we need a complete query operation.

We can use the `queries` static property on the `AppRoute` class that we defined in the previous chapter. This property is an object that represents one or more full GraphQL queries.

Add a static `queries` property to the `AppRoute` class definition in `js/app.js`:

```
class AppRoute extends Relay.Route {
  static routeName = 'App';
  static queries = {
    library: (Component) => Relay.QL `query QuotesLibrary {
      quotesLibrary {
        ${Component.getFragment('library')}
      }
    }
  }
}
```

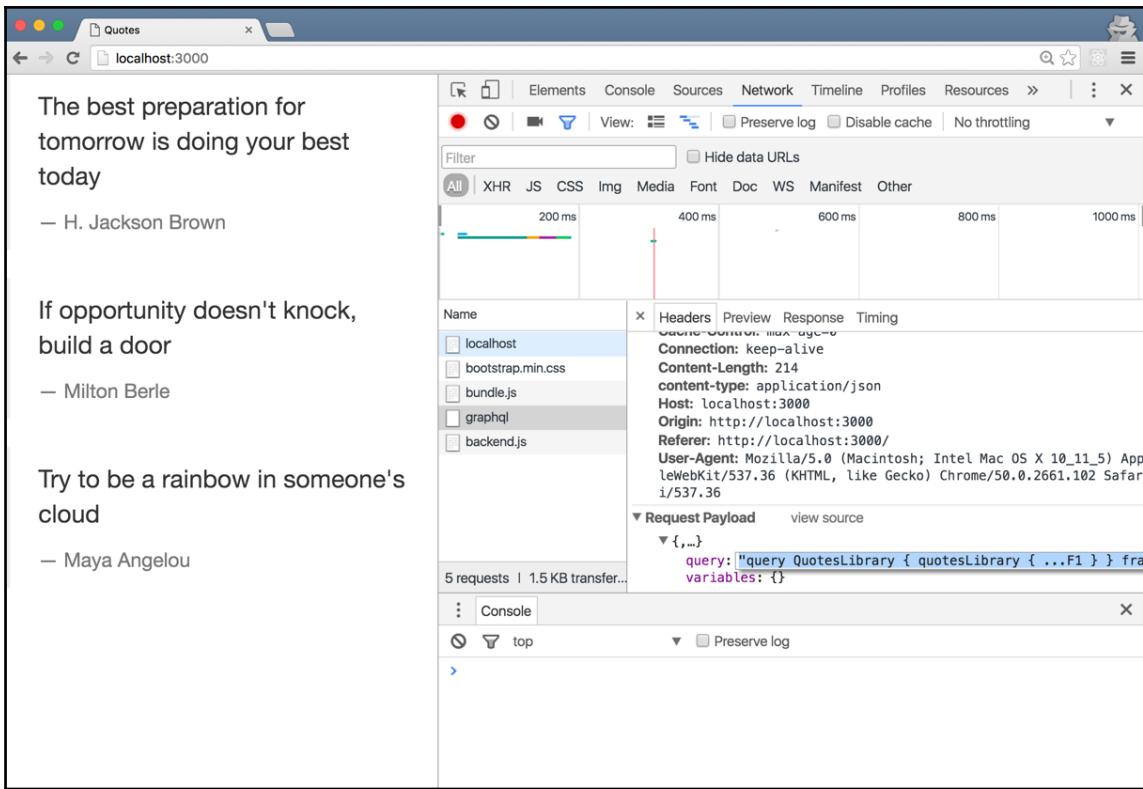
Every query is represented with a name, `library` for our single query in this example, and a function that receives one argument (`Component` in this example) and returns a full GraphQL query tagged with `Relay.QL`. The `Component` argument is what we passed to `RootContainer` when we defined it:

```
<Relay.RootContainer
  Component={QuotesLibrary}
  route={new AppRoute()}>
```

For our example, `Component` will be `QuotesLibrary` here, and we're constructing a GraphQL query that starts with the root field `quotesLibrary` and inside of that, uses the library named fragment defined by the `QuotesLibrary` component. When Relay constructs this full query, it will be the same full query that we've tested before:

```
{
  quotesLibrary {
    allQuotes {
      id
      text
      author
    }
  }
}
```

After a server restart and a webpack run, when we refresh the browser now, the application will work exactly as it did before, but we are now fetching the data through Relay (and not manually with a single query like we did before). To see how Relay constructs the query from the fragments that we defined, open the network tab in Chrome's dev tools and inspect the POST request for `/graphql`:



We can copy the text of the `query` value in the request payload, prettify, and execute it with GraphQL; we'll see the following:

```
query QuotesLibrary {
  quotesLibrary {
    ...F1
  }
}

fragment F0 on Quote {
  text
  author
  id
}

fragment F1 on QuotesLibrary {
  allQuotes {
    id
    ...F0
  }
}
```

```
}
```

```
}
```

The screenshot shows a GraphiQL interface with two panes. The left pane contains a GraphQL query:

```
1 query QuotesLibrary {
2   quotesLibrary {
3     ...F1
4   }
5 }
6
7 fragment F0 on Quote {
8   text
9   author
10  id
11 }
12
13 fragment F1 on QuotesLibrary {
14   allQuotes {
15     id
16     ...F0
17   }
18 }
```

The right pane shows the resulting JSON response:

```
{
  "data": {
    "quotesLibrary": {
      "allQuotes": [
        {
          "id": "574d12b94ec08043094840a2",
          "text": "The best preparation for tomorrow is doing your best today",
          "author": "H. Jackson Brown"
        },
        {
          "id": "574d12b94ec08043094840a3",
          "text": "If opportunity doesn't knock, build a door",
          "author": "Milton Berle"
        },
        {
          "id": "574d12b94ec08043094840a4",
          "text": "Try to be a rainbow in someone's cloud",
          "author": "Maya Angelou"
        }
      ]
    }
  }
}
```

Below the query in the left pane, there is a "QUERY VARIABLES" section.

Notice how Relay constructed the full query using the fragments from the various components, but it renamed the fragments (F0, F1).

```
*** #GitTag: chapter5-first-query-operation-with-relay ***
```

Relay's connection model

Relay's power becomes obvious when we have to work with a big dataset. When, for example, we have hundreds of inspirational quotes, a common practice to do on listing pages is to paginate them, for example, by showing 20 quotes per page. Relay's connection model will help us implement this pagination feature quickly and efficiently.

We'll first need to convert the `allQuotes` field into a Relay *connection* field; we'll rename it `quotesConnection` to make that clear to clients who want to use it.

Just like any other custom type, we need to define a type for this new `quotesConnection` field. However, we can use a Relay helper in this case to create the new type. In `schema/main.js`, right after we define the `QuoteType`, add this:

```
const { connectionType: QuotesConnectionType } =  
  connectionDefinitions({  
    name: 'Quote',  
    nodeType: QuoteType  
});
```

`QuotesConnectionType` will be the type that we can use to define the new `quotesConnection` field.

`connectionDefinitions` is a helper function available from the `graphql-relay` package, which we need to install first:

```
~/graphql-project $ npm install --save graphql-relay  
└── graphql-relay@0.4.2
```

Then, we can import the Relay utility helpers from this package; let's import all the connection-related helpers in `schema/main.js`:

```
const {  
  connectionDefinitions,  
  connectionArgs,  
  connectionFromArray,  
  connectionFromPromisedArray  
} = require('graphql-relay');
```

The purpose of these helpers, in summary, is as follows:

- `connectionDefinitions` is a function that takes a configuration object describing a custom type in a GraphQL schema and returns another object with helpful helpers related to the Relay connection model around the custom type.
- `connectionArgs` is an object that defines all the arguments that can be used with the connection model. Those arguments are: `first`, `last`, `after`, and `before`. We'll see how to use them shortly.

- `connectionFromArray` is a function that takes a normal array object and converts it into a connection object for Relay.
- `connectionFromPromisedArray` is a function that takes a promise, which returns an array object, and converts that promise into a connection object for Relay. This is the function we'd want to use with our `quotesConnection` because when we query MongoDB, we only have a promise and not an actual array.

The `connectionDefinitions` object we defined for the `QuoteType` has a `connectionType` property; this is the property we can use to define our `quotesConnection` field. We destructured that property as `QuotesConnectionType`.

We're now ready to define the new `quotesConnection` field on the `QuotesLibraryType` in `schema/main.js` (replacing the old `allQuotes`):

```
const QuotesLibraryType = new GraphQLObjectType({  
  name: 'QuotesLibrary',  
  fields: {  
    quotesConnection: {  
      type: QuotesConnectionType,  
      description: 'A list of the quotes in the database',  
      args: connectionArgs,  
      resolve: (_ , args, { db }) => connectionFromPromisedArray(  
        db.collection('quotes').find().toArray(),  
        args  
      )  
    }  
  }  
});
```

A few things to notice here:

- The `quotesConnection` field type is the `QuotesConnectionType` object that we generated from Relay's `connectionDefinitions` for the `QuoteType`. This is Relay's way of expressing that `quotesConnection` is a connection that represent nodes of type `Quote`.
- The `quotesConnection` `args` object is defined as `connectionArgs`, which we imported from Relay as well. This means we can use these `connectionArgs` on a `quotesConnection` field.

- Instead of returning the MongoDB promise in the `resolve()` function, we returned a `connectionFromPromisedArray` invocation on that promise, and passed in as the second argument the `args` object, which is now defined as `connectionArgs`.

That's all we need to change in the schema; our list of quotes is now a Relay connection with pagination features we can use out of the box. For example, to fetch the first two quote objects, we can use this query (after a server restart):

```
{  
  quotesLibrary {  
    quotesConnection(first: 2) {  
      edges {  
        node {  
          id  
          text  
          author  
        }  
      }  
    }  
  }  
}
```

Here's how our GraphQL server would now respond to this query:

```
{  
  "data": {  
    "quotesLibrary": {  
      "quotesConnection": {  
        "edges": [  
          {  
            "node": {  
              "id": "574d12b94ec08043094840a2",  
              "text": "The best preparation for tomorrow  
is doing your best today",  
              "author": "H. Jackson Brown"  
            }  
          },  
          {  
            "node": {  
              "id": "574d12b94ec08043094840a3",  
              "text": "If opportunity doesn't knock,  
build a door",  
              "author": "Milton Berle"  
            }  
          }  
        ]  
      }  
    }  
  }  
}
```

```
        }
    }
}
```

The screenshot shows a GraphiQL interface with the following query:

```
1 {  
2   quotesLibrary {  
3     quotesConnection(first: 2) {  
4       edges {  
5         node {  
6           id  
7           text  
8           author  
9         }  
10      }  
11    }  
12  }  
13}  
14
```

The results pane displays the JSON response from the GraphQL server:

```
{  
  "data": {  
    "quotesLibrary": {  
      "quotesConnection": {  
        "edges": [  
          {  
            "node": {  
              "id": "574d12b94ec08043094840a2",  
              "text": "The best preparation for tomorrow  
is doing your best today",  
              "author": "H. Jackson Brown"  
            }  
          },  
          {  
            "node": {  
              "id": "574d12b94ec08043094840a3",  
              "text": "If opportunity doesn't knock,  
build a door",  
              "author": "Milton Berle"  
            }  
          }  
        ]  
      }  
    }  
  }  
}
```

Below the results, there is a "QUERY VARIABLES" section.

Notice the new hierarchy for the query fields. Relay added an `edges` field and a `node` field. This new hierarchy allows us to ask for meta information related to the pagination set. For example, Relay automatically adds a unique `cursor` for every node and allows us to figure out whether the current page is the last one or not; here's a query to read both of these new meta pieces of information about the current page:

```
{
  quotesLibrary {
    quotesConnection(first: 1) {
      pageInfo {
        hasNextPage
      }
      edges {
        cursor
        node {
          id
```

```
        text
        author
    }
}
}
}
}
```

The server's response for that is as follows:

```
{
  "data": {
    "quotesLibrary": {
      "quotesConnection": {
        "pageInfo": {
          "hasNextPage": true
        },
        "edges": [
          {
            "cursor": "YXJyYXljb25uZWN0aW9uObjA=",
            "node": {
              "id": "574d12b94ec08043094840a2",
              "text": "The best preparation for tomorrow
                     is doing your best today",
              "author": "H. Jackson Brown"
            }
          }
        ]
      }
    }
  }
}
```

The screenshot shows a GraphiQL interface with a code editor on the left and a results panel on the right. The code editor contains the following GraphQL query:

```
1  {
2    quotesLibrary {
3      quotesConnection(first: 1) {
4        pageInfo {
5          hasNextPage
6        }
7        edges {
8          cursor
9          node {
10            id
11            text
12            author
13          }
14        }
15      }
16    }
17 }
```

The results panel displays the JSON response corresponding to the query:

```
{
  "data": {
    "quotesLibrary": {
      "quotesConnection": {
        "pageInfo": {
          "hasNextPage": true
        },
        "edges": [
          {
            "cursor": "YXJyYXljb25uZWN0aW9uOjA=",
            "node": {
              "id": "574d12b94ec08043094840a2",
              "text": "The best preparation for tomorrow is doing your best today",
              "author": "H. Jackson Brown"
            }
          }
        ]
      }
    }
}
```

At the bottom left of the interface, there is a "QUERY VARIABLES" section.

The connection arguments defined by Relay are:

- **first:** We can supply an integer X here to tell Relay to return only the first X nodes from the connection list
- **last:** We can supply an integer X here to tell Relay to return only the last X nodes from the connection list
- **after:** We can supply a cursor value C here to tell Relay to start the current page after the node represented by the cursor C
- **before:** We can supply a cursor value C here to tell Relay to end the current page right before the node represented by the cursor C

For example, if our page size is 20 and the last cursor we've seen is "YXJyYXljb25uZWN0aW9uOjI=", here's how to tell Relay to fetch the nodes for the next page:

```
{
  quotesLibrary {
    quotesConnection(
      first: 20,
      after: "YXJyYXljb25uZWN0aW9uOjI="
```

```
) {  
  edges {  
    node {  
      id  
      text  
      author  
    }  
  }  
}  
}  
}
```



The connection argument logic is executed on the server for the whole array of quotes returned from MongoDB; this is not efficient on a large scale. Instead, we should pass these connection arguments to MongoDB itself and convert the pagination logic into MongoDB operations. For example, to honor the first argument, in MongoDB we can do something like this:

```
db.collection('quotes').find()  
.limit(args.first).toArray()
```

This, however, breaks parts of the Relay connection model. For example, the `pageInfo` field will behave differently, unless we customize it to also read its result directly from MongoDB.

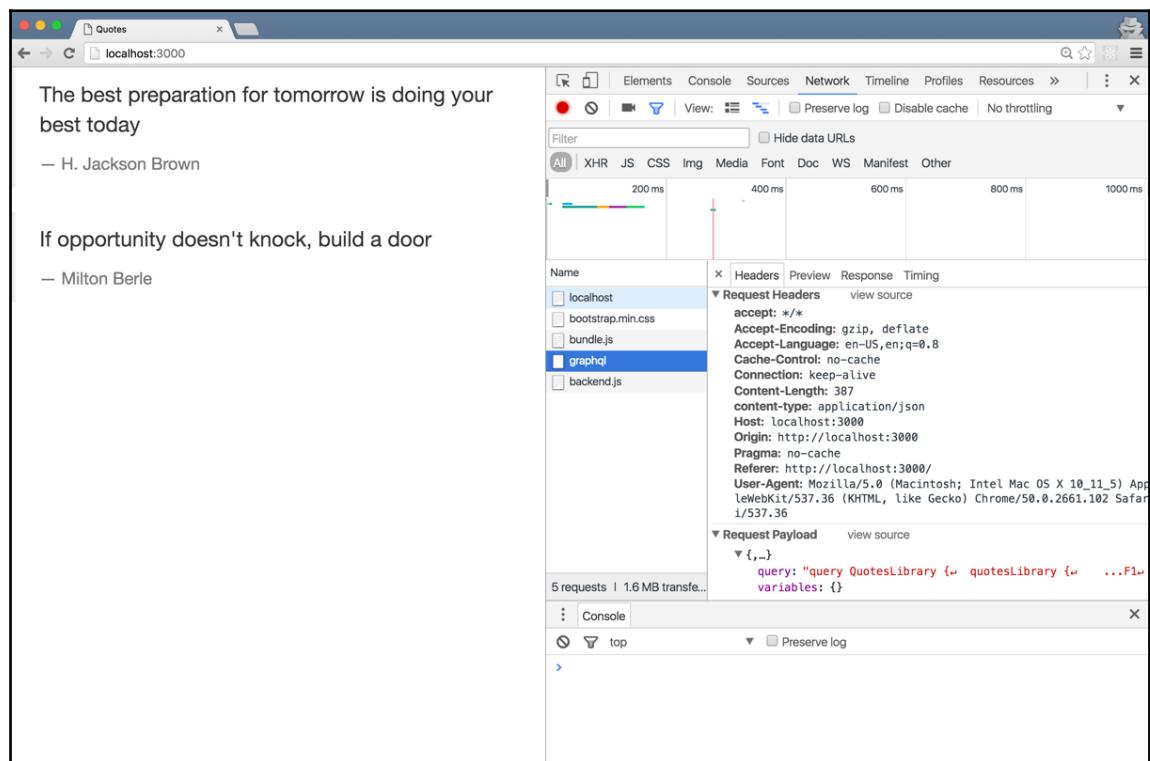
To make our React components work with this new connection model, we need to do a few modifications. First, the `library` fragment on the `QuotesLibrary` component (in `js/app.js`) will now have the new connection-based hierarchy:

```
QuotesLibrary = Relay.createContainer(QuotesLibrary, {  
  fragments: {  
    library: () => Relay.QL `  
      fragment on QuotesLibrary {  
        quotesConnection(first: 2) {  
          edges {  
            node {  
              id  
              ${Quote.getFragment('quote')}  
            }  
          }  
        }  
      }  
    }  
  }  
});
```

The `QuotesLibrary` component itself (in `js/app.js`) should also match this new hierarchy when it reads the data from its props:

```
class QuotesLibrary extends React.Component {
  render() {
    return (
      <div className="quotes-list">
        {this.props.library.quotesConnection.edges.map(edge =>
          <Quote key={edge.node.id} quote={edge.node} />
        )}
      </div>
    )
  }
}
```

When we refresh the browser now (after a webpack run and a server restart), we should see the first two inspirational quotes fetched via Relay through the new connection model we have for quotes:



Change the value of the connection `first` argument to 100. Doing the UI pagination actions is a matter of controlling this variable, along with another variable to control the `after` argument. We will see how to work with Relay variables in the next chapter.

*** #GitTag: chapter5-relay-connection-model ***

Summary

In this chapter, we started using GraphQL operations with Relay directly. Relay provides a plugin that allowed us to pre-process the queries against our GraphQL schema before using them in our client app. This allows Relay to have meta information about the different GraphQL operations, and that in turn gives Relay more power to control these operations.

We then matched React's component hierarchy in our GraphQL schema to make every React component able to represent its data requirement with a fragment on a GraphQL type that matches it. This allows for the co-location of the data requirements inside the components themselves in a way that separates each component's requirements from the requirements of the other components in the app. This allows us to maintain and test React's components independently.

We then implemented Relay's connection model for our list of quotes, which gave us pagination features out of the box. The connection model added edge and node logic to our GraphQL fields hierarchy, which allowed us to ask for extra information about the list of objects within different levels of the query.

6

Relay Variables and Object Identification

In this chapter, we'll explore how to work with Relay variables, then explore the object identification model in Relay using its Node interface. To work with Relay variables, we'll implement a search feature for our inspirational quotes application. To see the Node interface in action, we'll start implementing a *likes* counter on every quote.

The topics we'll cover in this chapter are as follows:

- Relay variables
- Relay global identifiers
- Relay's Node interface

Implementing search

Since we've limited the number of quotes we are displaying on the main page using Relay's connection model, let's give our users a way to search all quotes in our system. This will be a simple feature in the UI; we'll place a text box above all quotes, and when the user starts typing in that box, we'll refresh the displayed quotes based on a server search using the inputted search term.

This would not be an easy task without Relay. We need to construct a query using the input value from the user, fetch the records from GraphQL with AJAX, then change the state of our React app components based on the new data. We need to do this operation many times as the user types into the box. This is just the beginning of the story; to perform this efficiently, we need to manage the data. For example, basic caching is needed to not fetch duplicate responses. Imagine what would happen if the user types the word *do*, then types *or*, then erases the *or*; without caching, we will fetch the response for a query based on the word “do” twice.

Luckily, we have Relay to do the bulk of what is needed for this feature, but we still have a few things that we need to do first.

Adding a search feature to the GraphQL API

Before we can have Relay control a search feature, we need our GraphQL API to support a search mechanism in our list of quotes. We can use a field argument on the `quotesConnection` to pass a search term. For example:

```
{  
  quotesLibrary {  
    quotesConnection(searchTerm: "findme") {  
      edges {  
        node {  
          text  
        }  
      }  
    }  
  }  
}
```

The implementation of this on the server would require two changes:

- The `quotesConnection` field arguments should support a new `GraphQLString` argument `searchTerm`. Since the `quotesConnection` field already defines an argument object provided by Relay (`connectionArgs`), we'll need to extend that.
- When we resolve the `quotesConnection` field with a `db` promise, we'll need to modify the promise to only return the quotes filtered by the new `searchTerm` input. The syntax for that would depend on the database in use, but in MongoDB we'll need to use a regular expression.

Here's the new quotesConnection definition to support the searchTerm argument. This would replace the current QuotesLibraryType definition that we have in schema/main.js:

```
let connectionArgsWithSearch = connectionArgs;
connectionArgsWithSearch.searchTerm = { type: GraphQLString };

const QuotesLibraryType = new GraphQLObjectType({
  name: 'QuotesLibrary',
  fields: {
    quotesConnection: {
      type: QuotesConnectionType,
      description: 'A list of the quotes in the database',
      args: connectionArgsWithSearch,
      resolve: (_, args, { db }) => {
        let findParams = {};
        if (args.searchTerm) {
          findParams.text = new RegExp(args.searchTerm, 'i');
        }
        return connectionFromPromisedArray(
          db.collection('quotes').find(findParams).toArray(),
          args
        );
      }
    }
  }
});
```

We can test this right away with a valid search term (after a server restart):

```
{
  quotesLibrary {
    quotesConnection(searchTerm: "best") {
      edges {
        node {
          text
        }
      }
    }
  }
}
```

The screenshot shows the GraphiQL interface. On the left, a code editor displays a GraphQL query:

```
1 v {  
2 v   quotesLibrary {  
3 v     quotesConnection(searchTerm: "best") {  
4 v       edges {  
5 v         node {  
6 v           text  
7 v         }  
8 v       }  
9 v     }  
10 v   }  
11 }
```

On the right, the results pane shows the JSON response:

```
{  
  "data": {  
    "quotesLibrary": {  
      "quotesConnection": {  
        "edges": [  
          {  
            "node": {  
              "text": "The best preparation for tomorrow is doing  
your best today"  
            }  
          }  
        ]  
      }  
    }  
  }  
}
```

Below the code editor, the text "QUERY VARIABLES" is visible.

Note how the server responded with just one quote whose text matches the `searchTerm` value.

*** #GitTag: chapter6-adding-a-search-feature-to-the-graphql-api ***

Implementing the search feature in Relay

We'll need to read the search term needed from the UI somewhere. This part is not controlled by Relay—it's just a regular React component.

Adding a search form component

Let's create a `search-form.js` file under `js` and make it into a search form component:

```
import React from 'react';

class SearchForm extends React.Component {
  static propTypes = {
    searchAction: React.PropTypes.func.isRequired
  };

  handleChange = event => {
    this.props.searchAction(event.target.value);
  };
}
```

```
render() {
  return (
    <form className="navbar-form" role="search">
      <input type="text" className="form-control"
        placeholder="Search..." 
        onChange={this.handleChange} />
    </form>
  )
}

export default SearchForm;
```

This is a simple React component that has a single text input box. The component expects a `searchAction` prop that should be a function, and it will invoke this `searchAction` prop with whatever the user types in the box. The parent component will take it from there.



Note the use of `propTypes` to make sure the component receives the valid input it expects. `React.PropTypes.func` is one of many options we can use with React to validate component props. Chaining `isRequired` on `func` makes sure the component's `searchAction` property is not empty. React will give clear warnings when the validation rules are violated.

Using Relay variables

The parent component for our new `SearchForm` component will be `QuotesLibrary`. In there, we need to import `SearchForm` and include it somewhere within the rendered output. Let's first import it into `js/app.js`:

```
import { debounce } from 'lodash';

import SearchForm from './search-form';
```

I also imported the `debounce` function from the excellent `lodash` library (<https://lodash.com/>) so that we can properly debounce this search event and not send a query for every character while the user is typing.



A *debounced* function wraps an original function and gives it an acceptable delay. This allows us to group multiple sequential calls to the original function into a single one, and only invoke the original function once after the delay period. When we have an input box with an `onChange` event associated with a function `X`, a debounced version of that function would allow us to delay the execution of function `X` while the user is typing into the box, and only execute `X` once when the user stops typing.

We need to install the `lodash` dependency:

```
~/graphql-project $ npm install --save lodash
|__ lodash@4.14.2
```

We need to change the GraphQL fragment associated with `QuotesLibrary` to accept a `searchTerm` variable and pass it along to the GraphQL server. In `js/app.js`, modify the `Relay.createContainer` call for `QuotesLibrary` to the following:

```
QuotesLibrary = Relay.createContainer(QuotesLibrary, {
  initialVariables: {
    searchTerm: ''
  },
  fragments: {
    library: () => Relay.QL `fragment on QuotesLibrary {
      quotesConnection(first: 100, searchTerm: $searchTerm) {
        edges {
          node {
            id
            ${Quote.getFragment('quote')}
          }
        }
      }
    }
  }
});
```

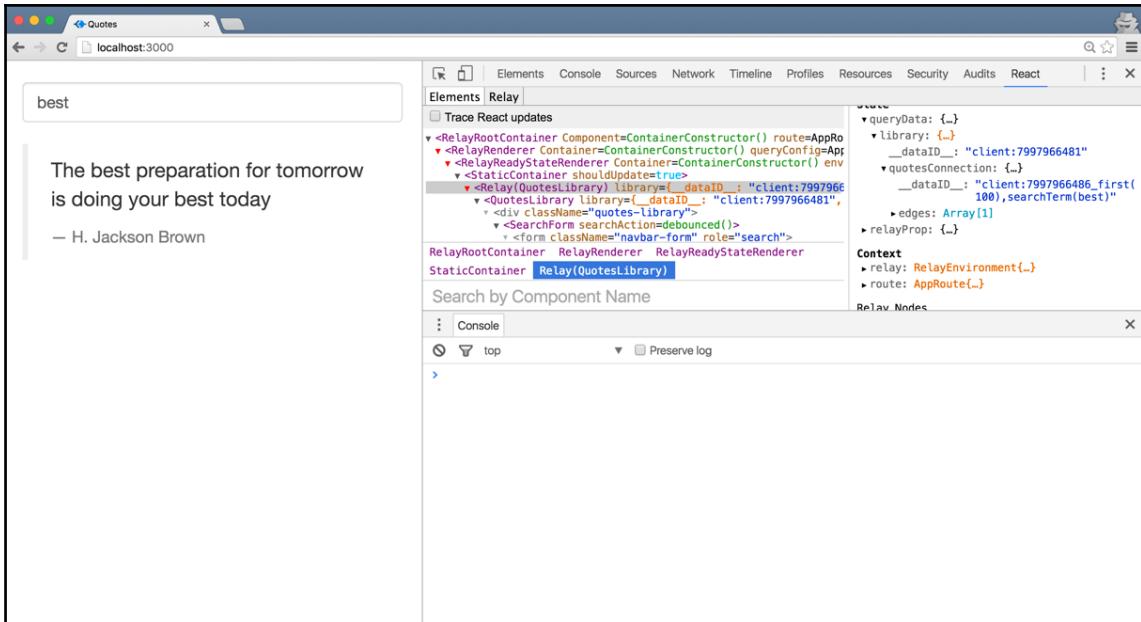
Note how we initialized the new `searchTerm` variable with an empty string. The MongoDB `find()` call that we customized to filter the quotes will ignore the empty `searchTerm` and the connection will resolve with the first 100 quotes.

Now all we need to do is instruct Relay to set this new `searchTerm` variable to the input we receive via `SearchForm`. Here's the new `QuotesLibrary` definition to implement the search feature (in `js/app.js`):

```
class QuotesLibrary extends React.Component {
  constructor(props) {
    super(props);
    this.search = debounce(this.search.bind(this), 300);
  }
  search(searchTerm) {
    this.props.relay.setVariables({searchTerm});
  }
  render() {
    return (
      <div className="quotes-library">
        <SearchForm searchAction={this.search} />
        <div className="quotes-list">
          {this.props.library.quotesConnection.edges.map(edge =>
            <Quote key={edge.node.id} quote={edge.node} />
          ))
        </div>
      </div>
    )
  }
}
```

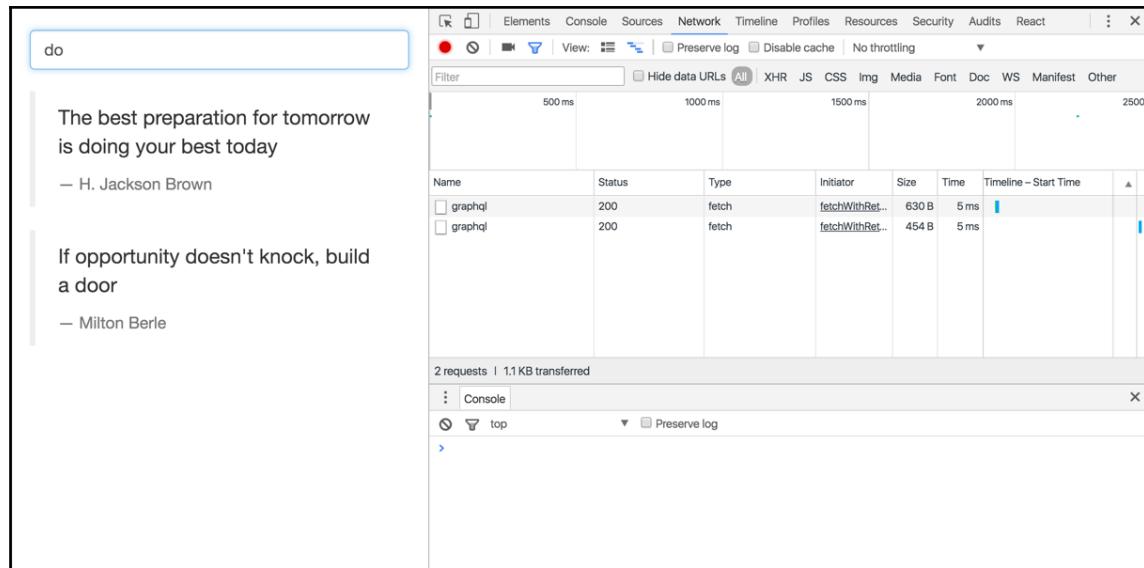
In the constructor call, we debounce the `search` function at 300 milliseconds. The `search` function receives the `searchTerm` from the `SearchForm` component and it just uses Relay's `setVariables` function to pass this input to the query. Relay will do the fetch call needed and update the data available for `this.props.library.quotesConnection`.

We can test the feature now, after a server restart and a webpack run:



Here's how you can see Relay's caching in action:

1. Refresh the application to load all quotes.
2. Open the **Network** tab in Chrome's developer tools and clear it.
3. Type "do" in the search box. You should see a single network request to /graphql. Two quotes are displayed in the filtered list.
4. Continue typing "or" in the search box. You should see another network request to /graphql. One quote is now displayed in the filtered list.
5. Erase "or" to get back to just "do" in the search box. The list will reflect the search for "do" (two quotes) without sending any more network requests to /graphql:



*** #GitTag: chapter6-implementing-the-search-feature-in-relay ***

Implementing likes

To demonstrate Relay's object identification design principles, let's introduce the famous Facebook *like* action. In the next chapter, we will implement the actual like action with a GraphQL mutation, but here we can start this feature out by displaying the number of likes for every quote. We'll use mock data for the likes count.

We'll first need to make the GraphQL API aware of this new field. We need to add it to the `QuoteType` object configuration, and we'll resolve it with a fake random number for now. Here's the new `QuoteType` definition that goes into `schema/main.js`:

```
const QuoteType = new GraphQLObjectType({
  name: 'Quote',
  fields: {
    id: {
      type: GraphQLString,
      resolve: obj => obj._id
    },
    text: { type: GraphQLString },
    author: { type: GraphQLString },
    likesCount: {
      type: GraphQLInt,
```

```
        resolve: () => Math.floor(10 * Math.random())
    }
}
});
```

We can see this new field in action by including the likesCount in any query (after a server restart):

```
{
  quotesLibrary {
    quotesConnection {
      edges {
        node {
          text
          likesCount
        }
      }
    }
  }
}
```

The screenshot shows a GraphiQL interface with a code editor on the left and a results panel on the right.

Code Editor (Left):

```
1 {  
2   quotesLibrary {  
3     quotesConnection {  
4       edges {  
5         node {  
6           text  
7           likesCount  
8         }  
9       }  
10     }  
11   }  
12 }
```

Results Panel (Right):

```
{  
  "data": {  
    "quotesLibrary": {  
      "quotesConnection": {  
        "edges": [  
          {  
            "node": {  
              "text": "The best preparation for tomorrow is  
doing your best today",  
              "likesCount": 9  
            }  
          },  
          {  
            "node": {  
              "text": "If opportunity doesn't knock, build  
a door",  
              "likesCount": 2  
            }  
          },  
          {  
            "node": {  
              "text": "Try to be a rainbow in someone's  
cloud",  
              "likesCount": 8  
            }  
          }  
        ]  
      }  
    }  
  }
```

We can now show this new `likesCount` field in our Relay application. However, we don't want to show the number of likes for every quote to keep the UI to a minimum; instead, we want to make that number show up when the user clicks or taps on a quote element.

We can use a GraphQL `@include` directive to control whether the `likesCount` should be fetched or not for each quote.

Here's the `Quote` component definition with this feature implemented (this is in `js/quote.js`):

```
class Quote extends React.Component {
  displayLikes() {
    if (!this.props.relay.variables.showLikes) {
      return null;
    }
    return (
      <div>
        {this.props.quote.likesCount} &nbsp;
        <span className="glyphicon glyphicon-thumbs-up">
        </span>
      </div>
    );
  }

  render() {
    return (
      <blockquote>
        <p>{this.props.quote.text}</p>
        <footer>{this.props.quote.author}</footer>
        {this.displayLikes()}
      </blockquote>
    );
  }
}
```

Here, we defined a `displayLink()` method to conditionally render the likes if `showLikes` is true. `showLikes` is the Relay variable we want to use to control the `@include` directive.

We can read Relay variables using `this.props.relay.variables`. When the `showLikes` variable is true, we're assuming that we have a `likesCount` field in our GraphQL query response.

Here's the Relay container definition for Quote (also in `js/quote.js`):

```
Quote = Relay.createContainer(Quote, {
  initialVariables: {
    showLikes: false
  },
  fragments: {
    quote: () => Relay.QL `
      fragment OneQuote on Quote {
        text
        author
        likesCount @include(if: $showLikes)
      }
    `
  }
});
```

`$showLikes` starts as false, and we always use it in an `@include` directive to control whether `likesCount` is to be included in the query response or not. This means on the page initial load, none of the `likesCount` values will be included in the data response.

Now, let's add an `onClick` event on every quote to set `showLikes` to true. We can accept this event on the container `blockquote` element for every quote using `onClick={this.showLikes}`:

```
// Adding the event handler on every quote
// This is the new Quote's render method in js/quote.js
render() {
  return (
    <blockquote onClick={this.showLikes}>
      <p>{this.props.quote.text}</p>
      <footer>{this.props.quote.author}</footer>
      {this.displayLikes()}
    </blockquote>
  );
}
```

This new `showLikes` method will just set the Relay variable to true. In `js/quote.js`, add this new method to the `Quote` component definition:

```
showLikes = () => {
  this.props.relay.setVariables({showLikes: true});
};
```

After a webpack run and a server restart, we can go ahead and test this now by clicking on a quote element. We will get a Relay error. Open the **Network** tab in Chrome to see that error:

The screenshot shows the Network tab in the Chrome DevTools. There are three failed requests to the endpoint `/graphql`, each with a status of 400 (Bad Request). The first request's Headers show a query for a quote with ID 57281cc43a88dba20500f9c2. The Request Payload for the first request is as follows:

```
query: "query OneQuote($id_0: ID!) { node(id: $id_0) { ...F0 } }"
variables: {id_0: "57281cc43a88dba20500f9c2"}
```

The JavaScript console below the Network tab shows several errors related to the failed requests, including warnings about fetchWithRetries and a warning about RelayPendingQueryTracker.js:242.

Before we inspect this error, notice how we have three network calls to `/graphql` that all have status 400, which is a `Bad Request`. Although we only clicked on a single quote once, this is because Relay has an automatic retry feature, which is available by default. If a network request fails during a Relay fetch process for any reason, Relay will retry it and give up after a number of retries (three by default, as we can see in this example).

To see the bad request query, inspect any of the three failed requests and look at its request payload's `query` value. Copy the content of the query, paste it in GraphQL, and prettify it:

```
query OneQuote($id_0: ID!) {
  node(id: $id_0) {
    ...F0
  }
}

fragment F0 on Quote {
  likesCount @include(if: true)
```

```
    id  
}
```

The screenshot shows the GraphiQL interface. On the left, a code editor displays a GraphQL query:

```
1  query OneQuote($id_0: ID!) {  
2     node(id: $id_0) {  
3         ...F0  
4     }  
5 }  
6  
7 fragment F0 on Quote {  
8     likesCount @include(if: true)  
9     id  
10 }  
11 }
```

On the right, the results pane shows a JSON object representing an error response:

```
{  
  "errors": [  
    {  
      "message": "Unknown type \\"ID\\".",  
      "locations": [  
        {  
          "line": 1,  
          "column": 23  
        }  
      ]  
    },  
    {  
      "message": "Cannot query field \\\"node\\\" on type \\\"RootQuery\\\".",  
      "locations": [  
        {  
          "line": 2,  
          "column": 3  
        }  
      ]  
    }  
  ]  
}
```

Below the code editor, a small "QUERY VARIABLES" section is visible.

Notice two things about this query:

- When we clicked on a quote, Relay constructed a new GraphQL query (named `OneQuote`), but it did not include the full fragment associated with each quote; there is no `text` or `author` in the query. Relay has a diffing algorithm behind the scene that diffs queries and fragments to figure out what incremental query to send to the server. In this example, within a quote fragment, Relay already asked for the `text` and `author` fields on the initial page load. Now, it needs the `likesCount` field which it does not have yet, so it constructs a query that only asks for the `likesCount` field.
- Since we want this `likesCount` field on a single quote, we need to send the ID of that quote to the GraphQL server so that the server can identify the quote object. We don't have an API to do that yet. However, Relay has a global syntax to fetch any object in the system using its unique ID. This is why we see a `node` field in this example with an `id` field argument. Relay uses this syntax to refetch more information about any node that it partially fetched before.

To get this node-based querying to work, we need to do two things:

- Give every object in our schema a unique global ID
- Implement Relay's Node Interface feature, make our custom types implement it, and use it to support a `node` field on the root query type

Relay's global ID

To implement the Node Interface on our server, we can use some of the `graphql-relay` package helpers. Namely, we need the `globalIdField()` function, the `fromGlobalId()` function, and the `nodeDefinitions()` function. Let's import all three from the package and add them to the `require('graphql-relay')` line that we have in `schema/main.js`:

```
const {
  globalIdField,
  fromGlobalId,
  nodeDefinitions,
  connectionDefinitions,
  connectionArgs,
  connectionFromArray,
  connectionFromPromisedArray
} = require('graphql-relay');
```

The `globalIdField()` function can be used to create a `GraphQLID` field that automatically resolves with a value that is globally unique across the whole schema. For example, we should give our `QuotesLibraryType` a global ID in case Relay wants to use that (this is in `schema/main.js`):

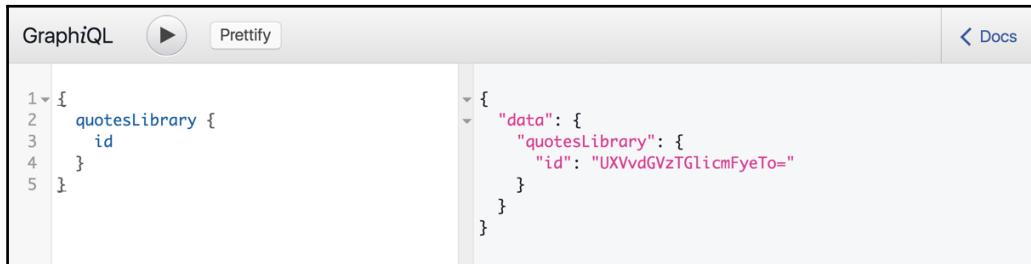
```
const QuotesLibraryType = new GraphQLObjectType({
  name: 'QuotesLibrary',
  fields: {
    id: globalIdField('QuotesLibrary'),
    quotesConnection: ...
  }
});
```

After a server restart, we can test this feature right away by asking about an `id` for the `quotesLibrary` root-level field:

```
{
  quotesLibrary {
    id
  }
}
```

The server will respond with the Relay-generated base64-encoded unique ID string:

```
{  
  "data": {  
    "quotesLibrary": {  
      "id": "UXVvdGVzTGlicmFyeTo="  
    }  
  }  
}
```



We can also convert the IDs we have for every quote to a Relay global ID using the same function. However, since this is not an ID for a single object in the whole schema, and we don't have a native `id` property on every quote object (it's `_id` in MongoDB), we can manually tell Relay how to read the `id` value using a *fetcher* function for the second argument of `globalIdField`. Replace the `id` field definition that we have currently in the `QuoteType` definition in `schema/main.js`:

```
const QuoteType = new GraphQLObjectType({  
  name: 'Quote',  
  fields: {  
    id: globalIdField('Quote', obj => obj._id),  
    text: { type: GraphQLString },  
    author: { type: GraphQLString },  
    likesCount: {  
      type: GraphQLInt,  
      resolve: () => Math.floor(10 * Math.random())  
    }  
  }  
});
```

Every quote `id` will now be unique in the whole schema, and not just in the `quotes` collection.

You can test that with the following query, after a server restart:

The screenshot shows the GraphiQL interface with a query editor and a results viewer. The query is:

```
1 {  
2   quotesLibrary {  
3     quotesConnection {  
4       edges {  
5         node {  
6           id  
7         }  
8       }  
9     }  
10   }  
11 }
```

The results pane displays the JSON response from the GraphQL server. It shows a nested structure where each edge has three nodes, each with a unique ID:

```
{  
  "data": {  
    "quotesLibrary": {  
      "quotesConnection": {  
        "edges": [  
          {  
            "node": {  
              "id": "UXVvdGU6NTcyODFjYzQzYTg4ZGJhMjA1MDBmOWMy"  
            }  
          },  
          {  
            "node": {  
              "id": "UXVvdGU6NTcyODFjYzQzYTg4ZGJhMjA1MDBmOWMz"  
            }  
          },  
          {  
            "node": {  
              "id": "UXVvdGU6NTcyODFjYzQzYTg4ZGJhMjA1MDBmOWM0"  
            }  
          }  
        ]  
      }  
    }  
  }  
}
```

At the bottom left, there is a "QUERY VARIABLES" section.

The Node interface

Now that every object in our schema has a globally unique ID, we are ready to implement the `node` field, which accepts a unique `id` as its argument and responds with the object that `id` identifies.

Every Relay global ID can be translated back to the two values we used to create it, its type name and its local ID within its collection. We can use Relay's `fromGlobalId()` function to do this translation.

For example, the following line shows a call to `fromGlobalId()` with one of our quotes node IDs:

```
fromGlobalId('UXVvdGU6NTcyODFjYzQzYTg4ZGJhMjA1MDBmOWMy')
```

This would return something like this:

```
{ type: 'Quote', id: '57281cc43a88dba20500f9c2' }
```

We can use these two values to locate the underlying object identified by this global ID.



There is also a `toGlobalId()` function that works exactly the opposite way to `fromGlobalId()`. This `toGlobalId()` is what the `globalIdField()` function uses as the resolver of the ID field it creates.

Relay provides a function called `nodeDefinitions` to help us with this task. We can use `nodeDefinitions` to create an interface that any object in our schema can implement.

The `nodeDefinitions()` function takes two arguments:

- The first argument is an ID fetcher function that we can use to resolve the global ID into its underlying object in our schema. This ID fetcher function can return a promise like any other `resolve()` function.
- The second argument is a type resolver function that we can use to map from the resolved underlying object to its GraphQL type.

Here's the `nodeDefinitions()` code that we need for our schema (this goes into `schema/main.js`):

```
const globalIdFetcher = (globalId, { db }) => {
  const { type, id } = fromGlobalId(globalId);
  switch (type) {
    case 'QuotesLibrary':
      // We only have one quote library
      return quotesLibrary;
    case 'Quote':
      return db.collection('quotes').findOne(ObjectID(id));
    default:
      return null;
  }
};

const globalTypeResolver = obj => obj.type || QuoteType;

const { nodeInterface, nodeField } = nodeDefinitions(
  globalIdFetcher,
  globalTypeResolver
);
```

The `globalIdFetcher` function receives the `globalId` value and the global context object from the executor. It uses `fromGlobalId` to read the type and local ID from the global ID, and then, using a simple switch statement, it returns the object identified by that ID. For a `QuotesLibrary` type, we only have a single `quotesLibrary` object in the schema, so we return that regardless of the ID. For a `Quote` type, we ask MongoDB about the quote object with the supplied ID and return the promise object itself.

`ObjectID` is a MongoDB client utility that we need to import with the following (in `schema/main.js`):

```
const { ObjectId } = require('mongodb');
```

The `globalTypeResolver` is simple: if the underlying object has a type, return it; otherwise, default to the `QuoteType`. This default type will work when `globalIdFetcher` returns a promise for the single quote object.

We can now call `nodeDefinitions` with both functions as arguments. It will return two objects for us, one is the `nodeInterface` object that we can use to make any type implement this new interface, and the other is a `nodeField` object that we can use to define the `node` field on the root-level of our GraphQL API. The `nodeField` will allow clients to query our GraphQL API using the `node` field with an `id` argument, and it will resolve with the `globalIdFetcher` returned value.

Let's first make our types implement the `nodeInterface` to allow Relay to refetch them through the `node` field. Add the `interfaces` property to both `QuoteType` and `QuotesLibraryType` in `schema/main.js`:

```
const QuoteType = new GraphQLObjectType({
  name: 'Quote',
  interfaces: [nodeInterface],
  // fields ...
});

const QuotesLibraryType = new GraphQLObjectType({
  name: 'QuotesLibrary',
  interfaces: [nodeInterface],
  // fields ...
});
```

Now add the `nodeField` type to the root-level query object as `node`:

```
const queryType = new GraphQLObjectType({
  name: 'RootQuery',
  fields: {
    node: nodeField,
    quotesLibrary: {
      type: QuotesLibraryType,
      description: 'The Quotes Library',
      resolve: () => quotesLibrary
    }
  }
});
```

We also need to make sure the `quotesLibrary` object has a `type` property so that it works for the `globalTypeResolver` function. Replace the `quotesLibrary` constant in `schema/main.js` with the following:

```
const quotesLibrary = { type: QuotesLibraryType };
```

Now, after a server restart, any client can ask for our schema objects using the Node interface. Here's a test query to demonstrate that (the ID value you need here will be different, take one from the previous query we tested):

```
{
  node(id: "UXVvdGU6NTcyODFjYzQzYTg4ZGJhMjA1MDBmOWMy") {
    ... on Quote {
      text
      author
    }
  }
}
```

Note how we use an inline fragment to ask about the sub-fields of a `node`. A `node` can be either a `Quote` or a `QuotesLibrary` in our example here. In this test query, we're asking the server to reply with `text` and `author` when the `node` is a `Quote` object.

The server's response here would be as follows:

A screenshot of a GraphiQL interface. On the left, a code editor shows a GraphQL query:

```
1 ↵ {  
2 ↵   node(id: "UXVvdGU6NTcyODFjYzQzYTg4ZGJhMjA1MDBmOWMy") {  
3 ↵     ... on Quote {  
4 ↵       text  
5 ↵       author  
6 ↵     }  
7 ↵   }  
8 ↵ }
```

On the right, the response is displayed in JSON format:

```
{  
  "data": {  
    "node": {  
      "text": "The best preparation for tomorrow  
is doing your best today",  
      "author": "H. Jackson Brown"  
    }  
  }  
}
```

Relay's generated fragment to read the `likesCount` when we click on a quote should work now. After a webpack run and a server restart, clicking on a quote element in the UI should show its fake `likesCount`:

A screenshot of a browser developer tools Network tab. The left pane shows a list of network requests:

- localhost
- bootstrap.min.css
- bundle.js
- graphql
- backend.js
- graphql (selected)
- glyphicon-halflings-regular.woff2

The right pane shows the details for the selected "graphql" request. The Headers tab displays:

```
accept: */*  
Accept-Encoding: gzip, deflate  
Accept-Language: en-US,en;q=0.8,ar;q=0.6  
Connection: keep-alive  
Content-Length: 288  
content-type: application/json  
Host: localhost:3000  
Origin: http://localhost:3000  
Referer: http://localhost:3000/  
User-Agent: Mozilla/5.0 (Macintosh; Intel Mac OS X 10_11_5) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/50.0.2661.102 Safari/537.36
```

The Request Payload tab shows the GraphQL query:

```
query OneQuote($id_0:ID!) {  
  node(id:$id_0) {  
    ...F0...  
  }  
}  
variables: {id_0: "UXVvdGU6NTcyODFjYzQzYTg4ZGJhMjA1MDBmOWMy"}
```

*** #GitTag: chapter6-implementing-likes ***

Summary

In this chapter, we implemented two new features for our inspirational quotes library. The first feature is a search box with type-ahead support, where we update the list of displayed quotes when the user types anything in the search box. We implemented this feature using a Relay variable that we passed to GraphQL. The second feature was a likes count that we now show for each quote when the user clicks on it. For this feature, we used another Relay variable to control a GraphQL directive and saw how Relay uses a `node` field to ask for any new information it needs. To enable reading through this `node` field, we implemented an interface on the GraphQL server, made our custom types implement it, and added the `node` field to the root query type. The interface maps between global object identifiers and their underlying objects. In the next chapter, we'll use Relay mutations to allow users to like a quote.

7

Relay Mutations

In this chapter, we will define a Relay-compliant GraphQL mutation and see how to invoke it in Relay. The mutation will allow users to like a quote. Once liked, the likes count in the UI will increase.

The topics we'll cover in this chapter are:

- Defining a Relay-compliant mutation
- Invoking mutations with Relay
- Relay's optimistic updates

Relay-compliant GraphQL mutations

In the previous chapter, we implemented the likes count feature and used mock random data to test it out. Let's now allow the users to click on the thumbs-up icon to like a quote and replace the mock random data we used before with actual data in the database.

First, we need to define a GraphQL mutation to record the like action. This mutation will take one input argument, the ID of the quote to be liked, and it will return the new number of likes for that quote.

Let's name this new mutation `thumbsUp`, and we'll need to add it to the root-level mutation type in `schema/main.js`:

```
const mutationType = new GraphQLObjectType({  
  name: 'RootMutation',  
  fields: {  
    thumbsUp: thumbsUpMutation  
  }  
});
```

```
const mySchema = new GraphQLSchema({
  query: queryType,
  mutation: mutationType
});
```

The `mutation` property on a GraphQL schema defines one or more mutations within the `fields` property of its configuration. Our `thumbsUp` mutation will be defined in a `thumbsUpMutation` function, which we'll need to define in `schema/main.js` right above the `mutationType` object:

```
const thumbsUpMutation = mutationWithClientMutationId({
  name: 'ThumbsUpMutation',
  inputFields: {
    quoteId: { type: GraphQLString }
  },
  outputFields: {
    quote: {
      type: QuoteType,
      resolve: obj => obj
    }
  },
  mutateAndGetPayload: (params, { db }) => {
    const { id } = fromGlobalId(params.quoteId);
    return Promise.resolve(
      db.collection('quotes').updateOne(
        { _id: ObjectID(id) },
        { $inc: { likesCount: 1 } }
      )
    ).then(result =>
      db.collection('quotes').findOne(ObjectID(id)))
  }
});
```

`mutationWithClientMutationId` is another helper function that we can import from `graphql-relay`. Add it to the `require('graphql-relay')` line in `schema/main.js`:

```
const {
  mutationWithClientMutationId,
  // other helpers
} = require('graphql-relay');
```

We can use `mutationWithClientMutationId` to:

- Define a single input structure for the mutation. All Relay-compliant mutations expect a single input field argument that wraps all the actual input values we want to pass to the mutation. In our case, the only input value we're passing to this mutation is `quoteId`, which is a string. The user will click the thumbs-up button on a single quote, and the server needs to identify that quote to increment its `likesCount` property.
- Define the output fields of the mutation. Those are all the fields we can read after the mutation is done. For our example, we are going to expose a single `quote` object to represent the quote that got liked in this mutation.
- Define the `mutateAndGetPayload` function which, as the name implies, will do two operations: it will invoke the actual mutation logic, and then it will return the payload to be exposed to the output fields of the mutation.

`mutateAndGetPayload` maps from the input fields to the output fields using the mutation operation. The first argument it receives is the list of the input parameters, which we can read to perform the mutation action. It also receives the global `context` object in case we want to access any of its properties (we do need the `db` property for this example).

`mutationWithClientMutationId` returns a field configuration object, which is what we used on the root-level mutation field, `thumbsUp`.

The `ThumbsUpMutation` takes a quote ID, increments the `likesCount` of that quote in the database, and then reads the quote object and outputs it back in the payload.

The code for the actual mutation logic is MongoDB-specific, but the main idea is that we use the `updateOne` function along with an `$inc` operation to increment `likesCount`, and once that is done successfully, we return the quote object in a promise.

The object we return from `mutateAndGetPayload` can be accessed within the output fields `resolve()` functions as the first argument. For our example, it'll be a promise that resolves with the quote.

We can now remove the mock random `likesCount` logic from the `QuoteType` and read the database property instead. Modify the `QuoteType` definition in `schema/main.js` to be:

```
const QuoteType = new GraphQLObjectType({
  name: 'Quote',
  interfaces: [nodeInterface],
  fields: {
    id: globalIdField('Quote', obj => obj._id),
    text: { type: GraphQLString },
    author: { type: GraphQLString },
    likesCount: {
      type: GraphQLInt,
      resolve: obj => obj.likesCount || 0
    }
  }
});
```

The `resolve` function here will return `likesCount` if it exists and default to 0 otherwise.

After a server restart, we can test our mutation now. In GraphiQL, test the following operation:

```
mutation ThumbsUp($input: ThumbsUpMutationInput!) {
  thumbsUp(input: $input) {
    clientMutationId
    quote {
      likesCount
    }
  }
}
```

Test it with the following query variables:

```
{
  "input": {
    "clientMutationId": "1",
    "quoteId": "UXVvdGU6NTcyODFjYzQzYTg4ZGJhMjA1MDBmOWMy"
  }
}
```

For the `quoteId` value, pick one of the values we've seen in the previous queries when we queried for all the quote IDs. The quote IDs depend on the data in your local database. If you use an invalid `quoteId` value, you should get a GraphQL error.

Here's a query to read the quote ID for the last quote in the connection:



```
{  
  quotesLibrary {  
    quotesConnection(last: 1) {  
      edges {  
        node {  
          id  
        }  
      }  
    }  
  }  
}
```

Here's what we should see when you invoke the mutation operation with a valid `quoteId` value:

The screenshot shows a GraphiQL interface with a query editor and a results panel. The query editor contains a mutation named `ThumbsUP` that updates a quote's likes count. The results panel shows the JSON response with the updated quote data. Below the editor, there is a "QUERY VARIABLES" section containing the mutation input variables.

```
GraphiQL Prettify Docs  
  
mutation ThumbsUP($input: ThumbsUpMutationInput!) {  
  thumbsUp(input: $input) {  
    clientMutationId  
    quote {  
      likesCount  
    }  
  }  
}  
  
mutation ThumbsUP($input: ThumbsUpMutationInput!) {  
  thumbsUp(input: $input) {  
    clientMutationId  
    quote {  
      likesCount  
    }  
  }  
}  
  
QUERY VARIABLES  


```
1 {
2 "input": {
3 "clientMutationId": "1",
4 "quoteId": "UXVvdGU6NTc0ZDEyYjk0ZWVmODA0MzA5NDg0MGE
5 }
6 }
7 }
```


```

You can execute this query a few times to see how the `likesCount` value is increasing with every run.

A couple of things to notice about this query:

- We defined the single argument input variable to be a required value of type `ThumbsUpMutationInput`. You can't invoke a mutation without an input object. The `ThumbsUpMutationInput` is the mutation input type that Relay automatically creates for every mutation using its name. This type gets created in the `mutationWithClientMutationId()` call.
- We included a `clientMutationId` value of 1 in the input and the server included the same field in the response of the mutation. This is a unique identifier in every Relay application which Relay uses to identify every mutation. Every time Relay needs to invoke a new mutation operation, it increments the application's `clientMutationId` value and uses that in the new mutation request. The `mutationWithClientMutationId()` call created this field for us for both the input and output of the mutation.

Read the `likesCount` field on the same quote object to make sure the increment was persisted to the database (note how we can use the node interface here):

The screenshot shows the GraphiQL interface. On the left, the query is pasted:

```
1 {  
2   node(id: "UXVvdGU6NTc0ZDEyYjk0ZWVmODA0MzA5NDg0MGEy")  
3     ... on Quote {  
4       text  
5       likesCount  
6     }  
7   }  
8 }  
9 |
```

On the right, the response is displayed:

```
{  
  "data": {  
    "node": {  
      "text": "The best preparation for tomorrow is doing  
your best today",  
      "likesCount": 3  
    }  
  }  
}
```

We are now ready to invoke the `thumbsUp` mutation field from within our Relay application:

*** GitTag: chapter7-relay-compliant-graphql-mutations ***

Relay.Mutation

We'll first need an event handler to invoke the mutation. We'll use a click handler on the thumbs-up icon.

In the `displayLikes()` method on the `Quote` component (in `js/Quote.js`), modify the span for the icon to include an `onClick` event:

```
displayLikes() {
  if (!this.props.relay.variables.showLikes) {
    return null;
  }
  return (
    <div>
      {this.props.quote.likesCount} &nbsp;
      <span className="glyphicon glyphicon-thumbs-up"
        onClick={this.thumbsUpClick}></span>
    </div>
  );
}
```

`thumbsUpClick` will simply instantiate a mutation class and pass it to `Relay.Store.commitUpdate`:

```
// In js/quote.js Quote component definition
thumbsUpClick = () => {
  Relay.Store.commitUpdate(
    new ThumbsUpMutation({
      quote: this.props.quote
    })
  );
};
```

The instantiated object from the `ThumbsUpMutation` class receives a single configuration object that represents the input that we want the mutation to have access to. Since we need to include the quote ID in the input of the mutation call, we can pass the whole `quote` object available on every `Quote` component prop.

We then pass this `ThumbsUpMutation` instance object to `Relay.Store.commitUpdate` to actually send the mutation request to the GraphQL server.

We need to import the `ThumbsUpMutation` class in `js/quote.js` where we used it:

```
import ThumbsUpMutation from './thumbs-up-mutation';
```

Now inside a `thumbs-up-mutation.js` file (under `js/`), we can define the class using:

```
import Relay from 'react-relay';

class ThumbsUpMutation extends Relay.Mutation {

  getMutation() {
    return Relay.QL `mutation {
      thumbsUp
    }`;
  }

  getVariables() {
    return {
      quoteId: this.props.quote.id
    };
  }

  getFatQuery() {
    return Relay.QL `fragment on ThumbsUpMutationPayload {
      quote {
        likesCount
      }
    }`;
  }

  getConfigs() {
    return [
      {
        type: 'FIELDS_CHANGE',
        fieldIDs: {
          quote: this.props.quote.id
        }
      }
    ];
  }
}

export default ThumbsUpMutation;
```



I did not use any folder structure for organizing the code in this book to keep things simple. This is fine for small projects, but once the list of files grows beyond a certain point, maintaining a project without a folder structure will be a challenge. Pick a folder structure and be consistent about using it. You can, for example, place all Relay mutations under a mutations folder in the `js` directory, or you can opt to create a folder per feature in your application. This advice also applies to the server side and the GraphQL schema which we wrote in a single file (`schema/main.js`) to keep things simple. In reality, you might want to have every custom GraphQL type in its own module, and group modules either by their operation type (query versus mutation versus subscription) or relation to a feature (the quotes collection versus the users collection), or a mixed approach for bigger schemas.

`ThumbsUpMutation` is a class that extends the `Relay.Mutation` class, and it requires four main getter functions:

- `getMutation()` is where we specify the name of the GraphQL mutation field that we want this mutation class to invoke. For our example, it's `thumbsUp`. As usual, we tag this GraphQL request with `Relay.QL`.
- `getVariables()` is where we define the structure of the single input object we want this mutation to use. For our example, we only need a `quoteId` variable, and we can read it using the `quote` property. This property holds what was passed to the `ThumbsUpMutation` instantiated object.
- `getFatQuery()` is where we represent, using a GraphQL fragment, everything in our data model that *could* change as a result of this mutation. For our simple example, we're only including the `likesCounts` field on a quote. Practically, multiple things could change as a result of a mutation. For example, if we're to later store the currently logged-in user (after we implement a login feature) along with the action of liking a quote, then this mutation would have two things in its fat query: the list of likers, and the total number of likes. Relay will not ask the server about the fat query as it is but instead it will intersect this query with a *tracked* query that represents the data our application currently uses. Relay will use the intersected query to ask the server for a response.
- `getConfigs()` is where we instruct Relay what to do with the response of the mutation request once it receives it. This could be an array of different operations; sometimes we want to add the response to a list, other times we want to merge the response with what we already have, and many other options. For our example, we want to change the `quote` object identified by the ID that we have. The `FIELDS_CHANGE` config type will do that for us.

The last thing we need to do to get this mutation working is to satisfy its data requirements. This mutation requires the presence of a quote id property, as we can see in `getVariables()` and `getConfigs()`. The `Quote` component did not require that `id` property so far. We can simply add the `id` field to the `Quote` component fragment Relay container, but that would not be the proper way to separate the requirements.

The proper way is to ask the mutation about its requirements and include them anywhere we use the mutation. For that, we can just use a simple fragment within the mutation (in `js/thumbs-up-mutation.js`):

```
class ThumbsUpMutation extends Relay.Mutation {  
  
  static fragments = {  
    quote: () => Relay.QL`  
      fragment on Quote {  
        id  
      }  
    };  
  
    // the getters  
  };
```

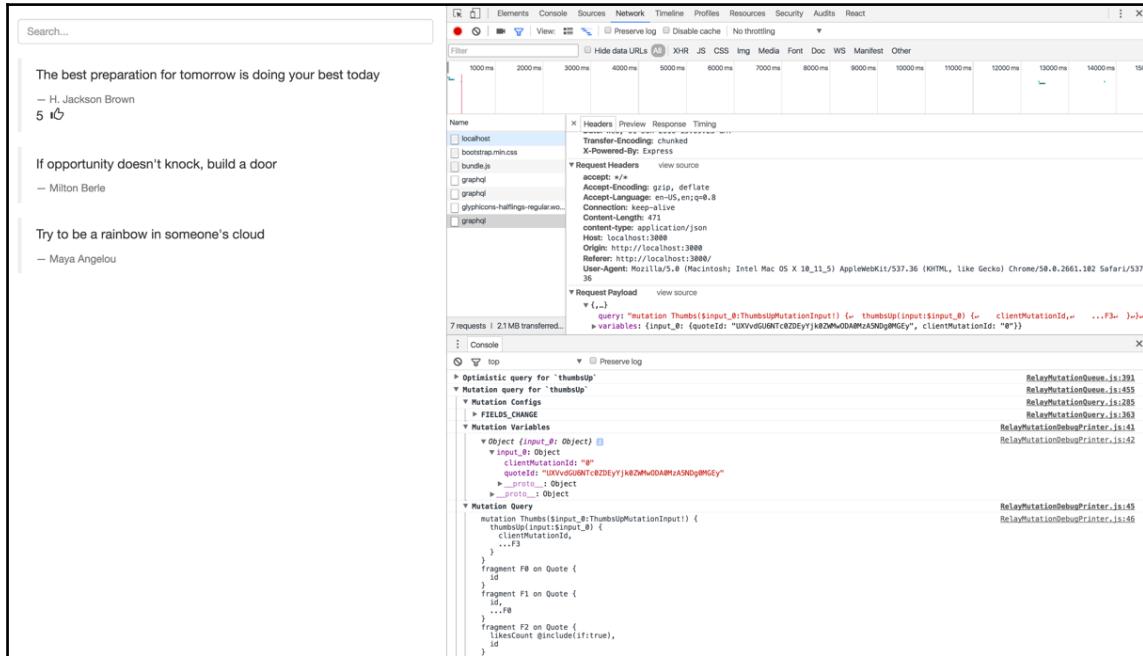
We only have one requirement, the quote requirement, where we specify that we need an `id` property on any `Quote` object on which this mutation will operate.

Within the fragment for the `Quote` component itself, we can include the mutation requirement using the same `getFragments` call we've used before.

In `js/quote.js`, add the following:

```
Quote = Relay.createContainer(Quote, {  
  initialVariables: {  
    showLikes: false  
  },  
  fragments: {  
    quote: () => Relay.QL`  
      fragment OneQuote on Quote {  
        ${ThumbsUpMutation.getFragment('quote')}  
        text  
        author  
        likesCount @include(if: $showLikes)  
      }  
    };  
  });
```

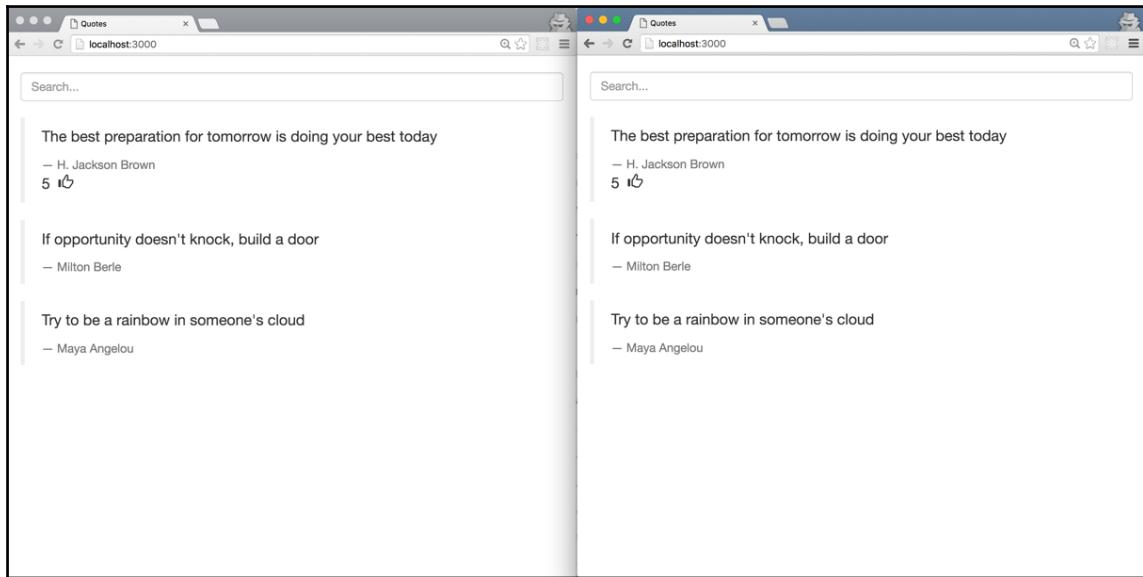
After a webpack run and a server restart, we can now test our mutation in the UI. Click on a quote object, then click on the thumbs-up icon. The likes count should immediately increment when the server responds with the new count:



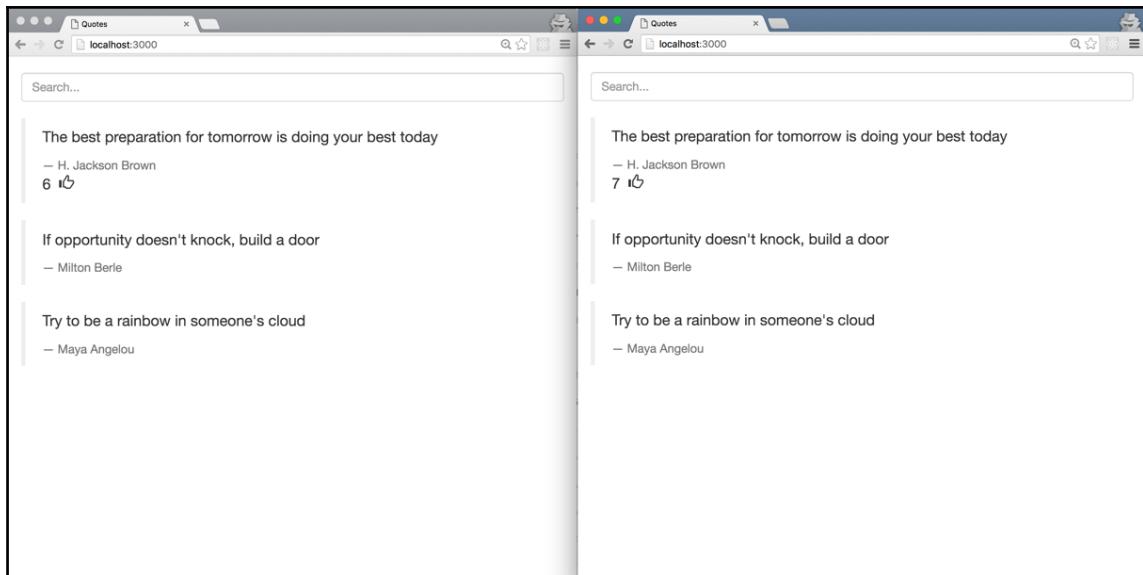
Note how the payload for the last network request is a mutation request. You can click the icon many times to like the quote many times. As a fun exercise, try to prevent the user from liking the quote multiple times, and display a filled icon when the user likes the quote displayed.

The likes counter does not simply get a +1 increment when we like a story, it actually uses the current likes count as it is in the database after we like the quote. For example, if, since the time we loaded up the `likesCount` the first time, someone else liked the quote, when we like it, we should see an increment of 2 in the count.

We can simulate a test of that by using two browser windows:



On initial load, the first quote in both windows for this example had a like counter of 5 likes. When we like the story in the first window, we should see the likes count going up to 6 there. When we like the story one more time in the second window, the likes count will get updated to 7 (from the original 5):



This is because every mutation is a write followed by a read, and we're reading the number of likes from the database in this mutation's read operation.

Relay mutations have many other features that you should explore. There are a lot of really good online documentation and examples about them. Take a look at all the mutation configuration options you can use in `getConfigs`, for example. There is a `NODE_DELETE` option to remove a node from a connection, and a `RANGE_ADD` option to append or prepend a node to a connection, and many others.

*** GitTag: `chapter7-relay-mutation` ***

Optimistic updates

One big built-in feature of Relay is its support for optimistic updates in the UI while we're waiting on a response from the server. Studies have shown that if the server took more than a few hundred milliseconds to respond, the UI will start to feel slow. Even if we have a super powered server that we know will never be slow, this UI lag will still be an issue for clients with limited internet connectivity.

Relay's mutations have a `getOptimisticResponse()` function that we can use to change the UI immediately after a mutation gets invoked. Here's the optimistic response we can use for our example. In `js/thumbs-up-mutation.js` `ThumbsUpMutation` class definition, add this:

```
getOptimisticResponse() {
  return {
    quote: {
      id: this.props.quote.id,
      likesCount: this.props.quote.likesCount + 1
    }
  };
}
```

The object that we return from this `getOptimisticResponse()` function will be used as the temporary response while the actual mutation is in flight. We're returning the current `likesCount` incremented by one. Just by using `this.props.quote.likesCount` in this function, we made this mutation depend on the existence of a `likesCount` property on `quote`. We can't assume its existence and should instead add it to the official requirements of the mutation.

Here's the complete definition of this mutation after we add this new requirement; this is in `js/thumbs-up-mutation.js`:

```
import Relay from 'react-relay';

class ThumbsUpMutation extends Relay.Mutation {

  static fragments = {
    quote: () => Relay.QL
      `fragment on Quote {
        id
        likesCount
      }`
  };

  getMutation() {
    return Relay.QL
      `mutation {
        thumbsUp
      }`;
  }

  getVariables() {
```

```
        return {
          quoteId: this.props.quote.id
        };
      }

      getFatQuery() {
        return Relay.QL `

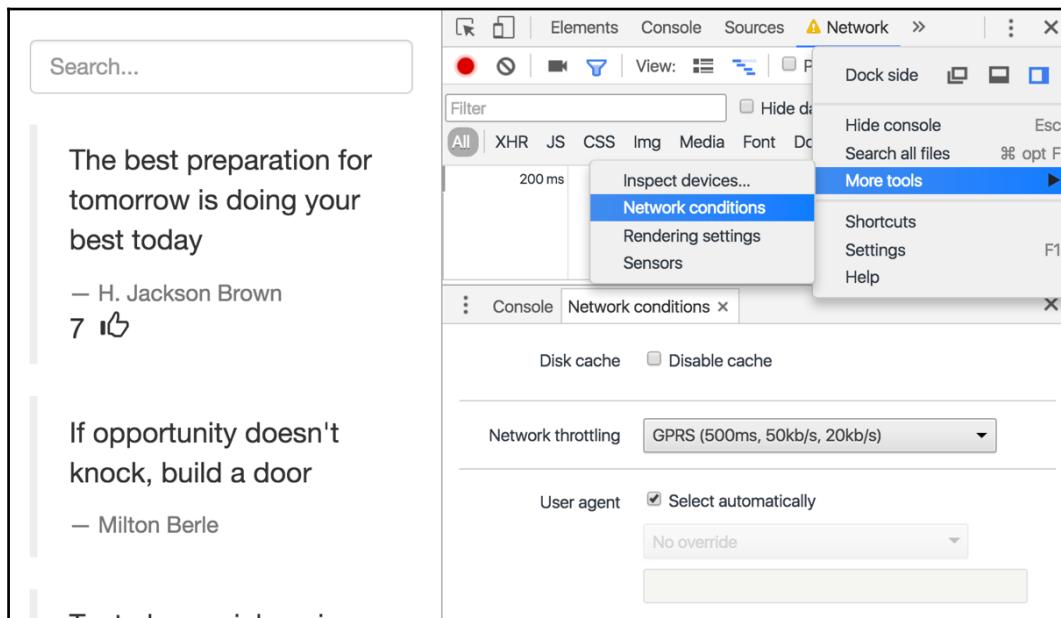
          fragment on ThumbsUpMutationPayload {
            quote {
              likesCount
            }
          }
        `;
      }

      getConfigs() {
        return [
          {
            type: 'FIELDS_CHANGE',
            fieldIDs: {
              quote: this.props.quote.id
            }
          }
        ];
      }

      getOptimisticResponse() {
        return {
          quote: {
            id: this.props.quote.id,
            likesCount: this.props.quote.likesCount + 1
          }
        };
      }
    }

    export default ThumbsUpMutation;
```

After a webpack run and a server restart, when we go back to the two windows test where both of them should have a likes count of 7 on the initial load, after we get 8 likes in the first window, the second window will also get an immediate 8 likes count right away using the optimistic response we defined. The 8 counter will get updated to 9 once we have the response from the server. On a locally hosted app, this test is hard to see, but you can simulate a slow client by throttling the network in Chrome. Pick the slowest connection available there:



*** GitTag: chapter7-optimistic-updates ***

Summary

In this chapter, we explored Relay-compliant GraphQL mutations. We learned how to define them using the `mutationWithClientId` helper function, and learned how that maps input fields to output fields using a mutation logic. We then learned how to invoke a mutation field in the UI by extending the `Relay.Mutation` class and calling `Relay.Store.commitUpdate` with a mutation object. We learned some features about Relay mutations such as the fat query, the mutator configurations, and the powerful built-in optimistic updates.

8

Deploying to the Cloud

In this chapter, we'll learn how to deploy the Relay Node application to the cloud. We will learn how to deploy to a cloud-based application platform (Heroku), and how to deploy to a virtual machine in the cloud (with Amazon Web Services). Most of the instructions in this chapter will apply to any Node application.

The topics we will cover in this chapter are:

- Deploying on EC2
- Deploying on Heroku

Preparing for deployment

Right now, we have a project that is working for us locally in development mode, and we want to take it to another server and run it there. The first thing that we need is a mechanism for us to copy the project files between our local machine and the remote server.

There are a few options we can go with here, but the most popular one is using Git source control. I am going to assume that you're familiar with Git and GitHub in this chapter, but if you're not, you can find great tutorials about Git at <https://git-scm.com/docs/gittutorial>, and GitHub has no shortage of resources about both Git and the GitHub platform. See <https://guides.github.com/>.



If you don't have Git installed on your system, it's time to install it. This link has good instructions on how to install Git on various platforms:
<https://git-scm.com/book/en/v2/Getting-Started-Installing-Git>.

Creating a GitHub repository

I have been using Git from the beginning for this project, which is what you should have been doing too. You get a lot of great benefits besides sharing the code base between multiple machines.

However, assuming that you have not made your `~/graphql-project` into a Git repository yet, here's what you need to do to make it into one:

```
~/graphql-project $ git init
Initialized empty Git repository in ~/graphql-project/.git/

~/graphql-project $ git status
On branch master

Initial commit

Untracked files:
  (use "git add <file>..." to include in what will be committed)

  .babelrc
  babelRelayPlugin.js
  cache/
  index.js
  js/
  node_modules/
  package.json
  public/
  schema/
  webpack.config.js

nothing added to commit but untracked files present (use "git add" to track)
```

We then need to add the files to the Git index. Before we do, we should ignore everything that gets generated when we run the application locally. To do this, in a `.gitignore` file, put the following lines:

```
node_modules/
public/bundle.js
public/bundle.js.map
cache/schema.json
```

`node_modules/` is where Node places all the dependencies when we run `npm install`, `bundle.js` and `bundle.js.map` both get generated by Webpack, and the `cache/schema.json` file is what we generate for the Relay Babel plugin. We should always add generated content to the ignored list and keep only the code we write in source control.

With those lines ignored, we can now `git add` all the files and `git commit` them as the first commit in our brand new repo:

```
~/graphql-project $ git add .

~/graphql-project $ git commit -m "Learn GraphQL/Relay Project"
[master (root-commit) f9b361f] Learn GraphQL/Relay Project
13 files changed, 479 insertions(+)
create mode 100644 .babelrc
create mode 100644 .gitignore
create mode 100644 babelRelayPlugin.js
create mode 100644 data/quotes
create mode 100644 index.js
create mode 100644 js/app.js
create mode 100644 js/quote.js
create mode 100644 js/search-form.js
create mode 100644 js/thumbs-up-mutation.js
create mode 100644 package.json
create mode 100644 public/index.html
create mode 100644 schema/main.js
create mode 100644 webpack.config.js
```

Note how the list of committed files did not include any of the generated files, thanks to the `.gitignore` file.

We can now share this Git repo with the world. The easiest way to do so is through GitHub. When you create a new repository on GitHub, it'll give you instructions on how to push your local repo there. The GitHub repo will have a location address like `https://github.com/username/repo-name.git`.

I pushed my copy to `https://github.com/edgecoders/graphql-project.git`, which is the address I'll be using in the following examples. You'll need to replace this address with your own, or you can use the copy I made. It's a public repository.

Deploying locally

Before we attempt any remote deploying steps, we should make sure our project works with a fresh copy of the local working directory. We can get a fresh copy by cloning the GitHub repo into a new test directory:

```
~ $ git clone https://github.com/edgecoders/graphql-project.git \
~/graphql-project-test

~ $ cd ~/graphql-project-test

~/graphql-project-test $ npm install

~/graphql-project-test $ node index.js

Connected to MongoDB server
{ Error: ENOENT: no such file or directory, open '~/graphql-project-
test/cache/schema.json'
  at Error (native)
  at Object.fs.openSync (fs.js:634:18)
  at Object.fs.writeFileSync (fs.js:1327:33)
  ...
  errno: -2,
  code: 'ENOENT',
  syscall: 'open',
  path: '~/graphql-project-test/cache/schema.json' }
Running Express.js on port 3000
```

While the Express.js server ran fine, the code we have to generate `cache/schema.json` failed because the `cache` directory does not exist.

We don't want to commit the `schema.json` file to Git but we should have an empty `cache` directory after a fresh checkout. Git ignores empty directories, but we can push an empty file in there:

Let's do the deployment changes in a new Git branch, `prod`:

```
~/graphql-project-test $ git checkout -b prod

~/graphql-project-test $ mkdir cache
```

Add an empty `.gitkeep` file under `cache`:

```
~/graphql-project-test $ touch cache/.gitkeep
```

Add the new `.gitkeep` file to Git:

```
~/graphql-project-test $ git add cache/.gitkeep

~/graphql-project-test $ git commit -m "Add the cache directory"
[prod 5d5d7e8] Add the cache directory
 1 file changed, 0 insertions(+), 0 deletions(-)
 create mode 100644 cache/.gitkeep
```

Let's see if this fixes the problem:

```
~/graphql-project-test $ node index.js
Connected to MongoDB server
Running Express.js on port 3000
Generated cached schema.json file
```

`index.js` has two responsibilities: starting the Express.js server, and generating the schema cache. We put these two tasks together to keep things simple, but it's time to split them up.

In a `generateSchemaCache.js` file on the root level, we can put only the code related to generating the JSON schema:

```
const fs = require('fs');
const path = require('path');
const { graphql } = require('graphql');
const { introspectionQuery } = require('graphql/utilities');
const mySchema = require('./schema/main');

graphql(mySchema, introspectionQuery)
  .then(result => {
    fs.writeFileSync(
      path.join(__dirname, 'cache/schema.json'),
      JSON.stringify(result, null, 2)
    );
    console.log('Generated cached schema.json file');
  })
  .catch(console.error);
```

In a `server.js` file on the root level, we'll keep the code related to starting the server:

```
const { MongoClient } = require('mongodb');
const assert = require('assert');
const graphqlHTTP = require('express-graphql');
const express = require('express');
```

```
const app = express();
app.use(express.static('public'));

const mySchema = require('./schema/main');
const MONGO_URL = 'mongodb://localhost:27017/test';

MongoClient.connect(MONGO_URL, (err, db) => {
  assert.equal(null, err);
  console.log('Connected to MongoDB server');

  app.use('/graphql', graphqlHTTP({
    schema: mySchema,
    context: { db },
    graphiql: true
  }));
}

app.listen(3000, () =>
  console.log('Running Express.js on port 3000')
);
});
```

Finally, we can now delete the `index.js` file.

Let's now write a few npm script tasks to work with these files. In `package.json`, under the `scripts` property, add the following:

- To generate the schema cache with `npm run generate-schema`, add this:

```
"generate-schema": "node generateSchemaCache.js"
```

- To build a production version of `bundle.js` with `webpack` using `npm run bundle`, add this:

```
"bundle": "NODE_ENV=production webpack -p --config webpack.config.prod.js"
```

- The `-p` is the production shortcut for the `webpack` command. It'll generate a minified `bundle.js`. Some libraries (React included) depend on `NODE_ENV` to generate a production-optimized build.
- For `NODE_ENV` to work with `webpack`, we need to add a plugin to the configuration file. We can make a separate `webpack.config.prod.js` root-level file for production-specific `webpack` configurations:

```
const path = require('path');
const webpack = require('webpack');
```

```
module.exports = {
  entry: './js/app.js',
  output: {
    path: path.join(__dirname, 'public'),
    filename: 'bundle.js'
  },
  module: {
    loaders: [
      {
        test: /\.js$/,
        exclude: /node_modules/,
        loader: 'babel-loader'
      }
    ]
  },
  plugins: [
    new webpack.DefinePlugin({
      'process.env': {
        'NODE_ENV': '"production"'
      }
    })
  ]
};
```

- To start the server with `npm start`, we'll need to add this line to the `scripts` section in `package.json`:

```
"start": "NODE_ENV=production node server.js"
```

We can test our scripts now. To prepare the schema and the JavaScript bundle file add the following:

```
~/graphql-project-test $ npm run generate-schema && npm run bundle
> node generateSchemaCache.js
Generated cached schema.json file
> NODE_ENV=production webpack -p --config webpack.config.prod.js

Hash: 3f76cb819c1c224d3588
Version: webpack 1.13.1
Time: 6803ms
 Asset   Size  Chunks             Chunk Names
bundle.js  484 kB      0  [emitted]  main
+ 464 hidden modules
```



You'll get a lot of logging information with the `-p` argument for Webpack; you can safely ignore those.

To start the server:

```
~/graphql-project-test $ npm start  
  
> NODE_ENV=production node server.js  
  
Connected to MongoDB server  
Running Express.js on port 3000
```

We now have our project running locally, in production mode, without any issues. If you inspect the size of the `bundle.js` file, you'll notice that it's now a much smaller file than what we had in development mode.

I'll go ahead and commit the new changes we made to the same `prod` branch:

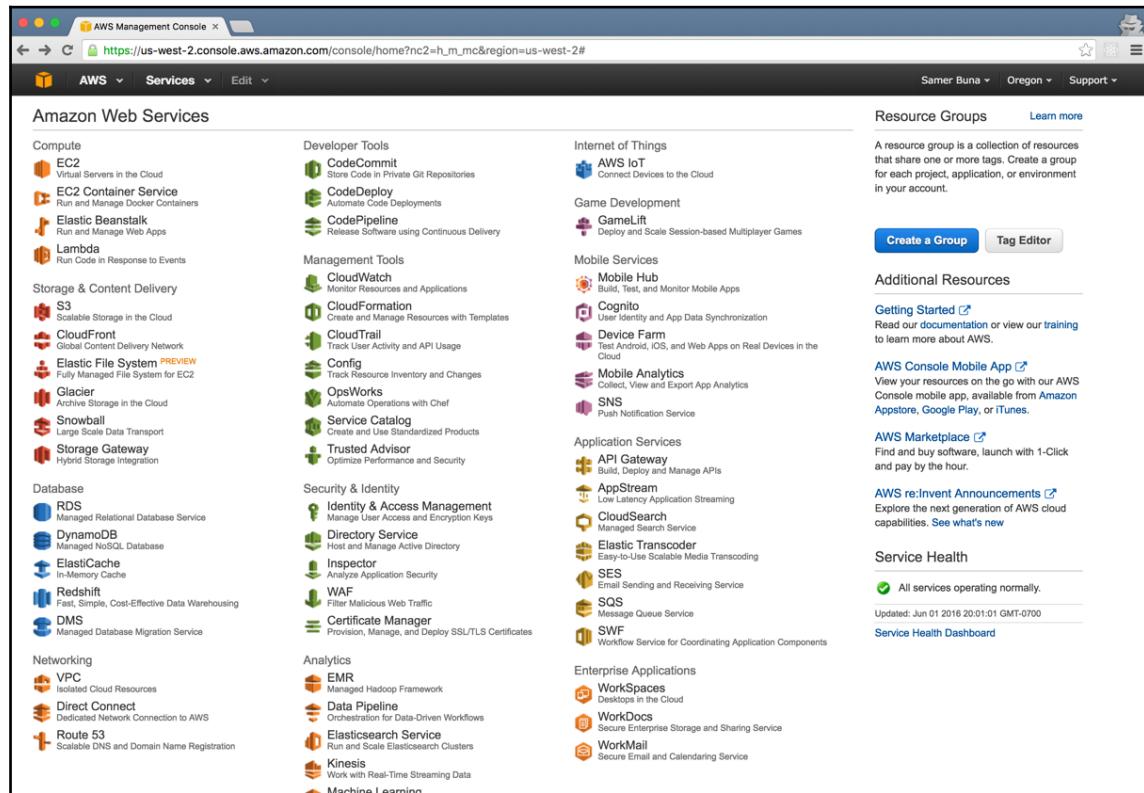
```
~/graphql-project-test $ git add .  
  
~/graphql-project-test $ git commit -m "Production Cleanup"  
[prod caaa8e0] Production Cleanup  
4 files changed, 44 insertions(+), 16 deletions(-)  
create mode 100644 generateSchemaCache.js  
rename index.js => server.js (59%)  
create mode 100644 webpack.config.prod.js
```

We're ready to go remote now. We'll first use **Amazon Web Services (AWS)** to create a **Virtual Private Server (VPS)** on their EC2 service and host our application there. After that, we'll see the simpler option of using the cloud-based application deployment platform, Heroku.

Deploying on EC2

First of all, you need to create an account on Amazon Web Services (AWS), which is hosted at: <https://aws.amazon.com/>.

Once you're all set with an active account, navigate to the AWS management console. It currently looks like this:



Pick the EC2 service, and then on that page, click the **Launch Instance** button:

The screenshot shows the AWS EC2 Management Console interface. On the left, there's a sidebar with various navigation links like EC2 Dashboard, Instances, Images, and Network & Security. The main content area has a heading 'Create Instance' with a sub-instruction: 'To start using Amazon EC2 you will want to launch a virtual server, known as an Amazon EC2 instance.' Below this is a large blue 'Launch Instance' button. To the right of the button, there's a section titled 'Service Health' which shows 'US West (Oregon)' status as 'operating normally'. Further down, there's a 'Scheduled Events' section indicating 'No events'. On the far right, there's an 'AWS Marketplace' section with a link to 'Tableau Server (10 users)'. At the bottom, there are links for 'Feedback', 'English', 'Privacy Policy', and 'Terms of Use'.

The screen will now prompt you to pick an Amazon Machine Image and an instance type. I usually go with an Ubuntu Server **t2.micro**. Click the **Review and Launch** button after that.

On the Review Instance Launch screen, you should see a **Security Groups** section with an SSH port 22 open by default. Since this machine will be a web server, we'll need to add port 80 too:

The screenshot shows the AWS EC2 Management Console interface for launching a new instance. The browser address bar shows the URL: https://us-west-2.console.aws.amazon.com/ec2/v2/home?region=us-west-2#LaunchInstanceWizard:.

The navigation bar at the top includes AWS Services, Edit, and tabs for 1. Choose AMI, 2. Choose Instance Type, 3. Configure Instance, 4. Add Storage, 5. Tag Instance, 6. Configure Security Group, and 7. Review. The 7. Review tab is currently selected.

Step 7: Review Instance Launch

Please review your instance launch details. You can go back to edit changes for each section. Click **Launch** to assign a key pair to your instance and complete the launch process.

AMI Details (Edit AMI)

Ubuntu Server 14.04 LTS (HVM), SSD Volume Type - ami-9abea4fb

Free tier eligible Ubuntu Server 14.04 LTS (HVM), EBS General Purpose (SSD) Volume Type. Support available from Canonical (<http://www.ubuntu.com/cloud/services>).
Root Device Type: ebs Virtualization type: hvm

Instance Type (Edit instance type)

Instance Type	ECUs	vCPUs	Memory (GiB)	Instance Storage (GB)	EBS-Optimized Available	Network Performance
t2.micro	Variable	1	1	EBS only	-	Low to Moderate

Security Groups (Edit security groups)

Security group name: launch-wizard-1
Description: launch-wizard-1 created 2016-06-01T20:21:53.679-07:00

Type (i)	Protocol (i)	Port Range (i)	Source (i)
SSH	TCP	22	0.0.0.0/0

Instance Details (Edit instance details)

Storage (Edit storage)

Buttons at the bottom: Cancel, Previous, Launch.

Footer: © 2008 - 2016, Amazon Web Services, Inc. or its affiliates. All rights reserved. Privacy Policy Terms of Use

Click on **Edit security groups**, then click on the **Add Rule** button, and pick **HTTP** in the selector under the **Type** column:

Step 6: Configure Security Group

A security group is a set of firewall rules that control the traffic for your instance. On this page, you can add rules to allow specific traffic to reach your instance. For example, if you want to set up a web server and allow Internet traffic to reach your instance, add rules that allow unrestricted access to the HTTP and HTTPS ports. You can create a new security group or select from an existing one below. [Learn more](#) about Amazon EC2 security groups.

Assign a security group: Create a new security group Select an existing security group

Security group name: launch-wizard-1

Description: launch-wizard-1 created 2016-06-01T20:21:53.667-07:00

Type	Protocol	Port Range	Source
SSH	TCP	22	Anywhere <input type="radio"/> 0.0.0.0/0 <input type="checkbox"/>
HTTP	TCP	80	Anywhere <input type="radio"/> 0.0.0.0/0 <input type="checkbox"/>

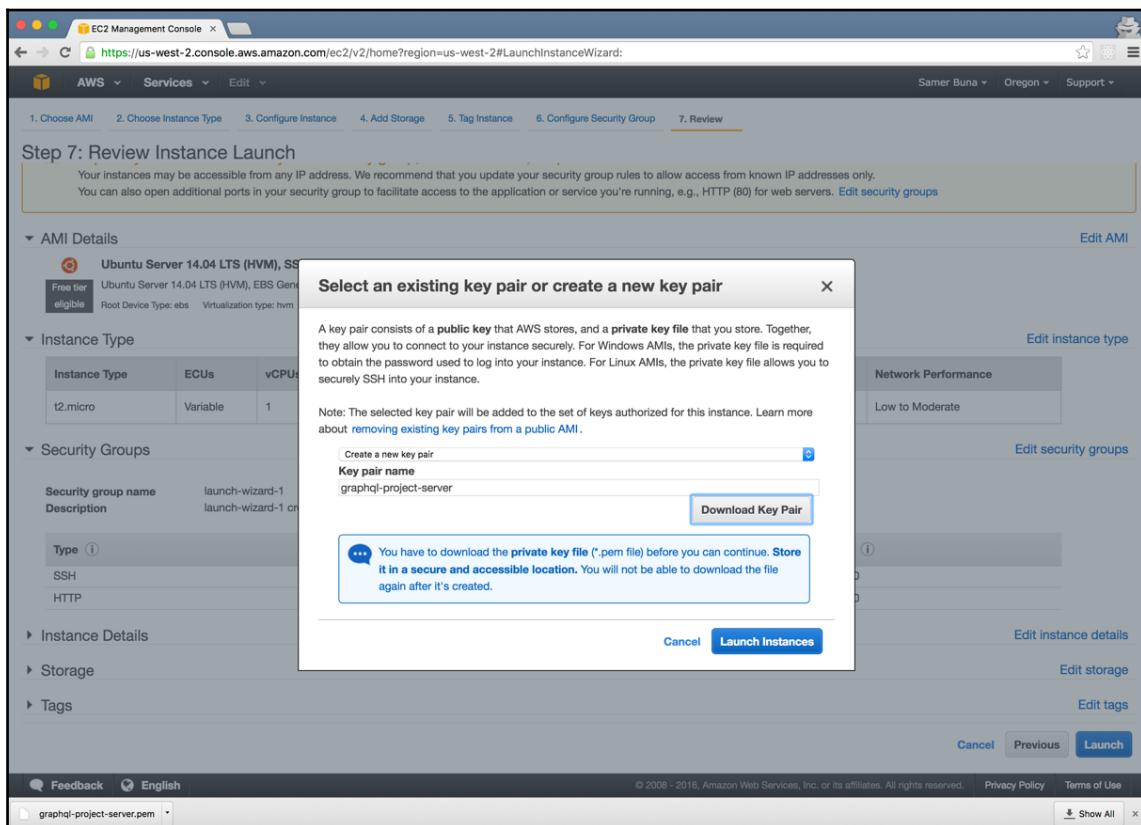
Add Rule

Warning: Rules with source of 0.0.0.0/0 allow all IP addresses to access your instance. We recommend setting security group rules to allow access from known IP addresses only.

Cancel Previous Review and Launch

Feedback English © 2008 - 2016, Amazon Web Services, Inc. or its affiliates. All rights reserved. Privacy Policy Terms of Use

We can go ahead and launch the instance now. The next screen should ask you to select an existing key pair or create a new one. If you have created a key pair before and have access to the .pem file, you can select that; otherwise, create a new key pair, give it a name like `graphql-project-server`, and download the new .pem file:



You'll need this `graphql-project-server.pem` file to access the EC2 instance. Don't lose it.

You can launch the instance now; this process will take some time. When the instance is ready, you'll see it running under the **Instances** screen. Take note of the Public IP that EC2 assigned to your instance; we'll need this IP to access the machine:

Name	Instance ID	Instance Type	Availability Zone	Instance State	Status Checks	Alarm Status	Public DNS
i-4fad6c94	t2.micro	us-west-2b	running	2/2 checks ...	None		ec2-54-191-153-32.i

We can now connect to this machine using an SSH terminal. You can see the exact instructions when you click the **Connect** button (with the machine selected).

Change the permission on the key we downloaded to make the file only readable by its owner:

```
~/Downloads $ chmod 400 graphql-project-server.pem
```

If you picked the Ubuntu Server machine, the SSH command you need is usually as follows:

```
~/Downloads $ ssh -i "graphql-project-server.pem" \
ubuntu@the.machine.ip.address
```

If you logged in successfully, your machine is ready.

To make sure the security group setting is correct, we can install apache2 using the following:

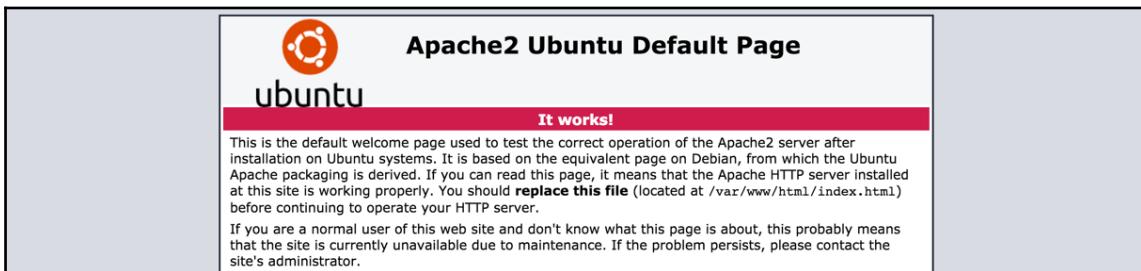
```
~ $ sudo apt install apache2
```

Make sure the apache2 process is running with this:

```
~ $ ps -ef | grep apache2
```

```
root      2286      1  0 03:53 ?          00:00:00 /usr/sbin/apache2 -k start
www-data  2289  2286  0 03:53 ?          00:00:00 /usr/sbin/apache2 -k start
www-data  2290  2286  0 03:53 ?          00:00:00 /usr/sbin/apache2 -k start
ubuntu    2412  1330  0 03:53 pts/0        00:00:00 grep -color=auto apache2
```

Then, go to `http://the.machine.ip.address/` in the browser and you should see the Apache default page there:



Installing MongoDB

We have two options to work with a remote MongoDB: a self-hosted install, or a database as a service platform. We'll explore the second option later in the chapter, but for an EC2 deployment, we'll install MongoDB and self-host it on the same server.

On an Ubuntu Server, we can simply `apt install` a package for MongoDB, but we first need to update the list of `apt` resources.

The full instructions can be found at <https://docs.mongodb.com/manual/tutorial/install-mongodb-on-ubuntu/>.

Here are the steps and commands that I needed:

1. Import the public key used by the package management system:

```
~ $ sudo apt-key adv --keyserver hkp://keyserver.ubuntu.com:80  
--recv EA312927
```

2. Create an apt resource file for MongoDB:

```
~ $ echo "deb http://repo.mongodb.org/apt/ubuntu trusty/  
mongodb-org/3.2 multiverse" | sudo tee /etc/apt/sources.list.d  
/mongodb-org-3.2.list
```

3. Reload the local package database:

```
~ $ sudo apt-get update
```

4. Install the MongoDB packages:

```
~ $ sudo apt-get install -y mongodb-org
```

To test that we have a successful MongoDB installation, we can use the `mongo` CLI:

```
~ $ mongo
MongoDB shell version: 3.2.8
connecting to: test
Welcome to the MongoDB shell.
>
```

While we're in the `mongo` interface, we can seed the database with some quotes:

```
> db.createCollection("quotes")
{ "ok" : 1 }

> db.quotes.insertMany([
...   {
...     text: "The best preparation for tomorrow
...           is doing your best today",
...     author: "H. Jackson Brown"
...   },
...   {
...     text: "If opportunity doesn't knock, build a door",
...     author: "Milton Berle"
...   },
...   {
...     text: "Try to be a rainbow in someone's cloud",
...     author: "Maya Angelou"
...   },
... ])
```

Running the Node.js server

Just like we used NVM in Chapter 1, *An Introduction to GraphQL and Relay*, to install Node.js locally, we can use it here too:

```
~ $ curl -o-
https://raw.githubusercontent.com/creationix/nvm/v0.31.4/install.sh | bash
```

Then, to install the latest Node.js, exit and reconnect to the server (this should load the nvm environment), then run the following:

```
~ $ nvm install node
```

We also need to install git on this server to be able to clone our project:

```
~ $ sudo apt install git
```

Then, to clone the project from GitHub, use the following:

```
~ $ git clone https://github.com/edgecoders/graphql-project.git
~ $ cd ~/graphql-project/
```

We've made some changes to the prod branch in the original repo. We'll need to checkout that branch:

```
~/graphql-project $ git checkout prod
Branch prod set up to track remote branch prod from origin.
Switched to a new branch 'prod'
```

Now we can npm install all the dependencies:

```
~/graphql-project $ npm install
```

If everything worked correctly, we should be able to generate the schema and bundle, and run the server, now using our npm scripts:

```
~/graphql-project $ npm run generate-schema
~/graphql-project $ npm run bundle
~/graphql-project $ npm start
```

However, while a direct run of the node command in the foreground works fine for development, in production, we need to manage the running node process in the background. Furthermore, if something stops this process for any reason, we need to attempt an auto-start. For that, we need to use a node process manager.

I've had a lot of success with the pm2 manager. It's very simple to use:

```
~/graphql-project $ npm install -g pm2
```

Then, to start a pm2 process based on the npm start script, we need to do the following:

```
~/graphql-project $ pm2 start npm -- start
[PM2] Starting npm in fork_mode (1 instance)
[PM2] Done.
```

The server is now running in the background on port 3000, and we have apache running on port 80. We can configure apache on port 80 as a proxy to our node server on port 3000 using the proxy_http apache module:

```
~/graphql-project $ sudo a2enmod proxy proxy_http
```

Change the content of /etc/apache2/sites-enabled/000-default.conf to the following:

```
<VirtualHost *:80>
    ServerName site.com
    ServerAlias www.site.com
    ProxyRequests off

    <Proxy *>
        Order deny,allow
        Allow from all
    </Proxy>

    <Location />
        ProxyPass http://localhost:3000/
        ProxyPassReverse http://localhost:3000/
    </Location>
</VirtualHost>
```

Restart the apache2 service (using the command `sudo service apache2 restart`) and go to `http://the.machine.ip.address/`, and you should see the application up and running in the cloud now.

This is just the beginning of the story on AWS. What we did here is a minimal deployment that suits a small application, but it will not scale easily for big ones. Here are a few things that we need to improve to set this application on a path for growth:

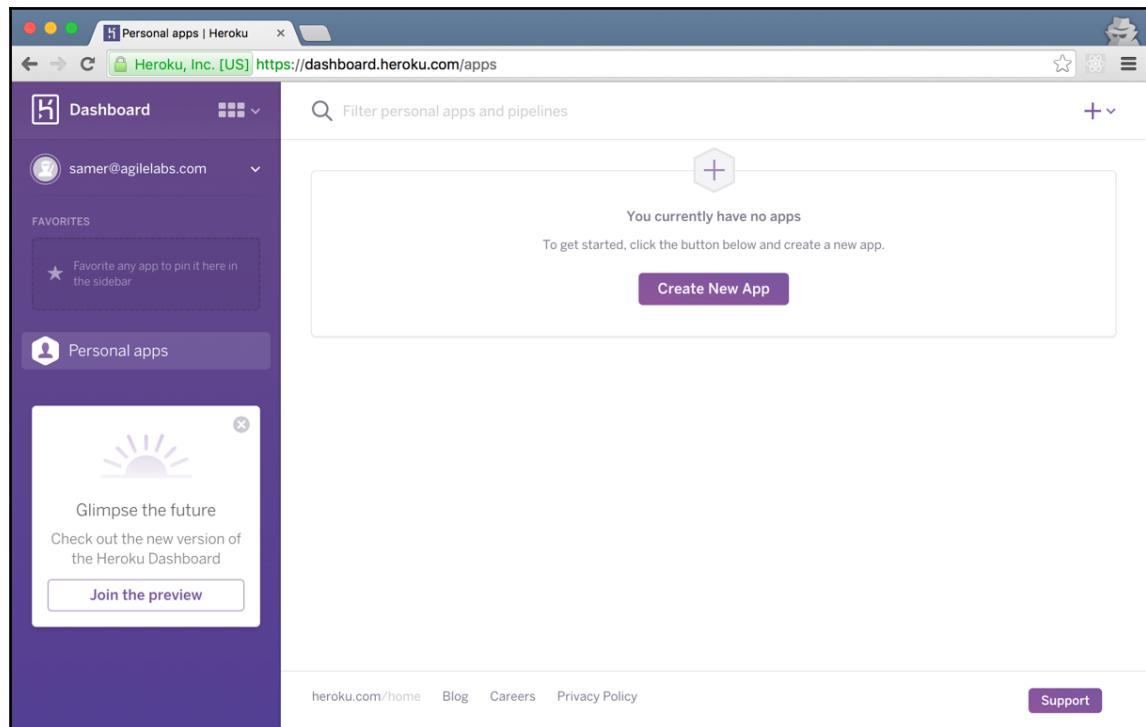
- Separate the application into three different sections. One section will be for the database cluster, the second section will be for the GraphQL/Express server process, and the third one will be for the frontend application. Every section can have many nodes that run independently and should be controlled by a load balancer and cache layer.
- Come up with an easier way to deploy new changes to all running nodes. Without a deployment strategy, we'll need to access each node for each deployment. There are a lot of tools to automate deployment and those should be utilized on an EC2 infrastructure.
- Come up with a plan to maintain the servers for things like security updates and resource usage. Servers that start misbehaving should be taken out of rotations right away, and if all nodes are busy, a new node should be automatically created and put in rotation.
- Enable a log aggregation service and monitor and analyze the central logs.

These are just a few points to get you thinking. AWS has other great services that can help with these concerns. For example, take a look at AWS *Elastic Beanstalk*, which supports Node.js.

EC2 and Beanstalk are very different services. EC2 offers Infrastructure as a Service (IaaS), while Beanstalk offers a Platform as a Service (PaaS). Heroku is another PaaS option, and it is specialized in that domain. An IaaS gives us a base that we can use to build things on top of. A PaaS, on the other hand, gives us an environment where we can just push our code and magically watch it come to life. IaaS has more power and flexibility, while PaaS offers an easier way to maintain, secure, and scale applications.

Deploying on Heroku

We'll need to create an account on <https://www.heroku.com/> first, and make sure you can log in to your dashboard, which currently looks like this:



You can use Heroku's dashboard to create and manage your applications, but there is also a Heroku command-line interface (CLI) utility that allows you to do the same operations right from the command line. The CLI should be easier to follow here.

To set up your local workstation with the Heroku CLI, you need to install the Heroku **Toolbelt** from <https://toolbelt.heroku.com/>.

The Heroku Toolbelt is available on all major platforms; just download it and run it. When the installer is done, open a terminal window and run the `heroku` command:

```
~/graphql-project-test $ heroku
heroku-cli: Installing CLI... 21.02MB/21.02MB
Enter your Heroku credentials.
Email: samer@agilelabs.com
Password (typing will be hidden):
Logged in as samer@agilelabs.com

      Add apps to this dashboard by favoriting them with heroku
apps:favorites:add
See all add-ons with heroku addons
See all apps with heroku apps --all

See other CLI commands with heroku help
```

The command will first install the latest Heroku CLI, and then ask you to enter your Heroku credentials.

Once logged in to the CLI, we can create a Heroku application. Run this command in the `graphql-project-test` local directory:

```
~/graphql-project-test $ heroku create
Creating app... done, pacific-reaches-64379
https://pacific-reaches-64379.herokuapp.com/ |
https://git.heroku.com/pacific-reaches-64379.git
```

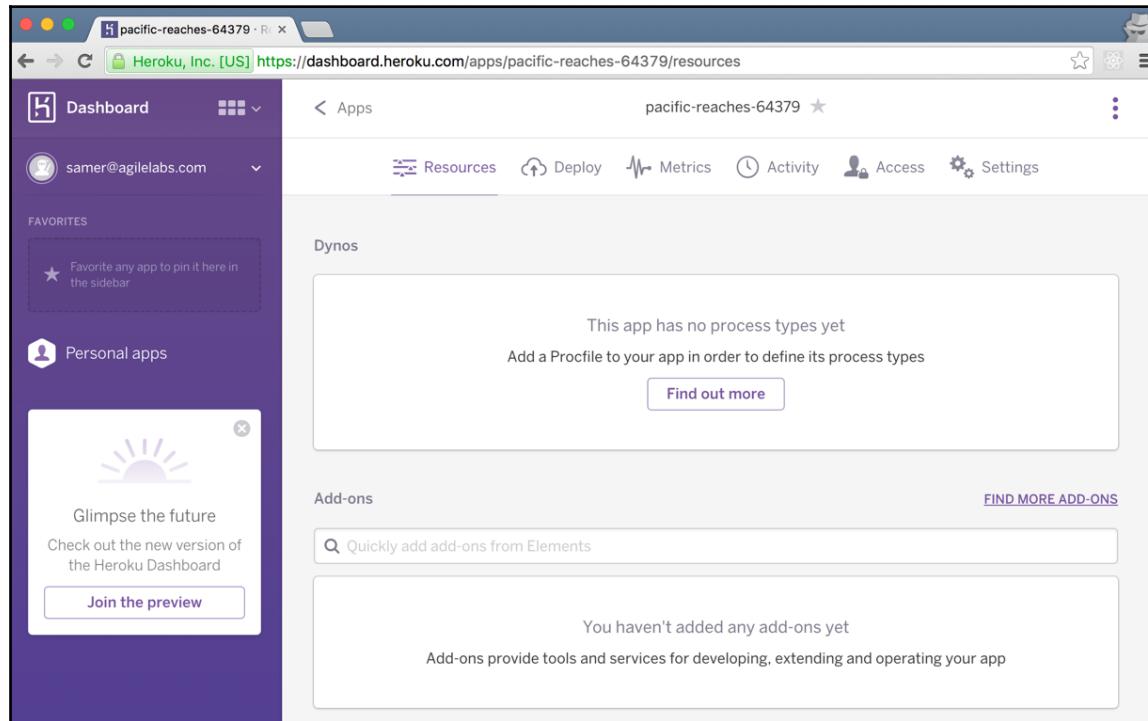
The CLI creates an application and generates a name for it. Mine was `pacific-reaches-64379`. After creating the application, the CLI associates a remote Git repository with your local one:

```
~/graphql-project-test $ git remote -v
heroku https://git.heroku.com/pacific-reaches-64379.git (fetch)
heroku https://git.heroku.com/pacific-reaches-64379.git (push)
origin https://github.com/edgecoders/graphql-project.git (fetch)
origin http://github.com/edgecoders/graphql-project.git (push)
```

The local repository has two remotes now, one at Heroku and the other at GitHub.

Heroku has many deployment methods; using a Heroku-managed Git repository is one of them, and it's the most popular.

At this point, if you reload your `https://www.heroku.com/` dashboard, you should see this newly created application in there. If you go to the application's page, you can see all the options that you can manage in the UI:



We'll continue working on the `prod` branch to add the changes required for Heroku:

```
~/graphql-project-test $ git status
On branch prod
Your branch is up-to-date with 'origin/prod'.
nothing to commit, working directory clean
```

The easiest way to test a Heroku application is to do so locally, using the `heroku local` command. We'll use this command to troubleshoot what we need to fix locally, before pushing the code.

To simulate a Heroku environment, I stopped the local MongoDB, and removed all generated files from the local repo:

```
~/graphql-project $ git clean -fxd
Removing cache/schema.json
Removing node_modules/
Removing public/bundle.js
Removing public/bundle.js.map
```

When the code is pushed to Heroku, the platform will perform the `npm install` step. Knowing that, we can perform this step locally, too. However, we also need the `schema.json` and `bundle.js` files. The `npm install` command will not generate those.

We can use the special `npm postinstall` script for that. This script gets automatically run when Heroku is done with the `npm install` command.

Here's the scripts section of `package.json` with the necessary `postinstall` task:

```
"scripts": {
  "generate-schema": "node generateSchemaCache.js",
  "bundle": "NODE_ENV=production webpack -p --config
    webpack.config.prod.js",
  "postinstall": "npm run generate-schema && npm run bundle",
  "start": "NODE_ENV=production node server.js"
},
```

When we run `npm install` now, npm will first install all the dependencies in `node_modules/`, and it will then execute the `postinstall` script and generate both the `schema.json` file and the production `bundle.js` file:

```
~/graphql-project-test $ npm install
...
> npm run generate-schema && npm run bundle
> node generateSchemaCache.js
Generated cached schema.json file
> NODE_ENV=production webpack -p --config webpack.config.prod.js
Hash: fe12498764cd49f048de
```

```
Version: webpack 1.13.1
Time: 9100ms
    Asset      Size  Chunks             Chunk Names
  bundle.js  484 kB        0  [emitted]  main
+ 461 hidden modules
```

Let's now see the first problem heroku local will run into:

```
~/graphql-project-test $ heroku local
[WARN] ENOENT: no such file or directory, open 'Procfile'
[OKAY] package.json file found - trying 'npm start'
[WARN] No ENV file found
8:58:12 AM web.1 | > graphql-project@1.0.0 start /Users/samer/graphql-
project-test
8:58:12 AM web.1 | > NODE_ENV=production node server.js
8:58:12 AM web.1 | /Users/samer/graphql-project-
test/node_modules/mongodb/lib/mongo_client.js:201
8:58:12 AM web.1 |           throw err
8:58:12 AM web.1 |           ^
8:58:12 AM web.1 |   AssertionError: null == { MongoError: failed to connect
to server [localhost:27017] on first connect
8:58:12 AM web.1 |       at <anonymous> (/Users/samer/graphql-project-t
8:58:12 AM web.1 |       at mongodb.MongoClient.connect
(/Users/samer/graphql-project-test/server.js:13:10)
8:58:12 AM web.1 |       at /Users/samer/graphql-project-
test/node_modules/mongodb/lib/mongo_client.js:198:11
8:58:12 AM web.1 |       at _combinedTickCallback
(internal/process/next_tick.js:67:7)
8:58:12 AM web.1 |       at process._tickCallback
(internal/process/next_tick.js:98:9)
8:58:12 AM web.1 | npm
```

It looks like heroku local is correctly executing the npm start script. However, when the server.js file tries to connect to a local MongoDB, it errors out because we stopped it.

We need a MongoDB for this application, and we can't use the local one on Heroku.

Cloud-hosted MongoDB

We've seen how to use a self-hosted MongoDB instance on EC2. Let's now explore the alternative option of using a cloud-hosted MongoDB service such as <https://www.compose.com/> or <https://mlab.com/>. This option is actually much easier to do and it's also a lot easier to maintain than the self-hosted option. Some of these cloud-hosted services even offer a free option (up to a certain usage size).

Once you create an account and create a database instance, you'll get the MongoDB URL that you can use in the Express.js app to replace the local URL that we've been using.

The MongoDB URL has this form:

```
mongodb://<dbuser>:<dbpassword>@<host>:<port>/<db-name>
```

On Heroku, we can obtain a database instance from <https://mlab.com/> using the third-party add-on system that Heroku offers. We can use the following command to add the free sandbox option:

```
~/graphql-project-test $ heroku addons:create mongolab:sandbox

Creating mongolab-rugged-10755... done, (free)
Adding mongolab-rugged-10755 to pacific-reaches-64379... done
Setting MONGODB_URI and restarting pacific-reaches-64379... done, v3
Welcome to mLab. Your new subscription is being created and will be
available shortly. Please consult the mLab Add-on Admin UI to check on its
progress.
Use `heroku addons:docs mongolab` to view documentation.
```

You might need to verify your Heroku account to be able to create add-ons.

This command creates a MongoDB cloud-based instance and sets a Heroku remote configuration variable to point to it. We can see all the Heroku remote configuration variables with the following:

```
~/graphql-project-test $ heroku config

==== pacific-reaches-64379 Config Vars
MONGODB_URI: mongodb://CREDS@HOST.mlab.com:PORT/DBNAME
```

This new MONGODB_URI variable can be accessed in the code using `process.env.MONGODB_URI`. We'll need to replace the hardcoded one we used before for the local database. In `server.js`, delete the line that sets the old `MONGO_URL` constant, and use `process.env.MONGODB_URI` in the connect call.

We should also remove the hardcoded port 3000. Heroku needs control over this. The port will be assigned during the deploy process and can be accessed in the code using `process.env.PORT`. In `server.js` add the following:

```
MongoClient.connect(process.env.MONGODB_URI, (err, db) => {
  // ...
  app.listen(process.env.PORT, () =>
    console.log(`Running Express.js on port ${process.env.PORT}`)
  );
});
```

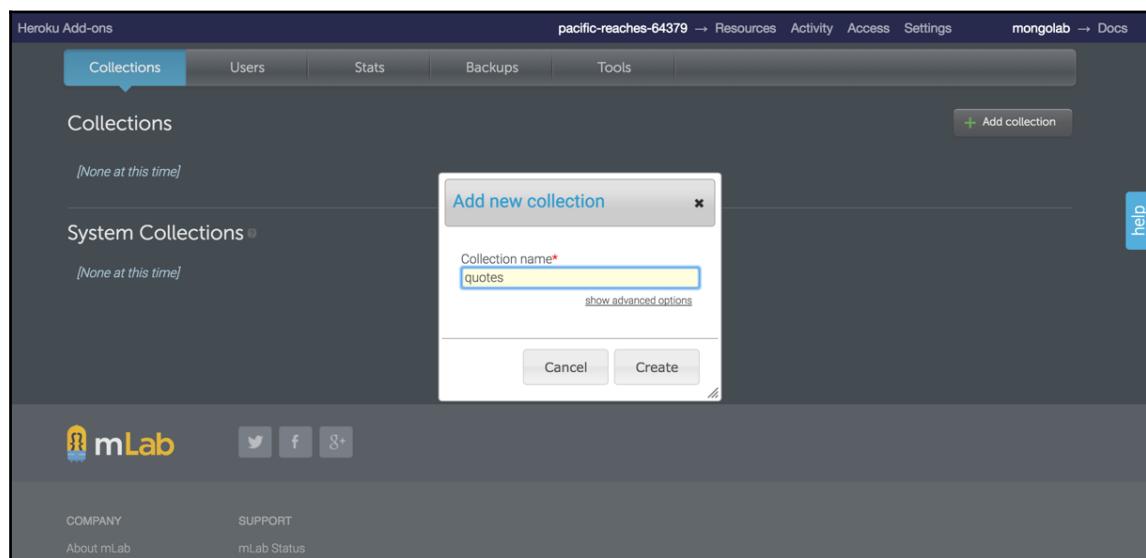
The `heroku local` command does not read the remotely-set configuration variables, but it does read any variables we set in a `.env` file. Using the following command, we can store a copy of the remote variables in a local `.env` file:

```
~/graphql-project-test $ heroku config -s >> .env
```

To access the MongoDB instance associated with this application, we can use the following command:

```
~/graphql-project-test $ heroku addons:open mongolab
```

This will open a window in your browser to a GUI for the MongoDB instance; we can use this UI to control the data we want in our quotes collection. First, create the quotes collection:



Then, use the quotes collection to insert a few quotes objects using the **Add Document** button. In the **Create Document** screen, every document is a JSON string:

The screenshot shows the 'Create document' page for the 'quotes' collection in the 'pacific-reaches-64379' database. The document content is:

```
1 {
2   "text": "The best preparation for tomorrow is doing your best today",
3   "author": "H. Jackson Brown"
4 }
```

At the bottom, there are three buttons: 'Cancel and go back', 'Create and go back', and 'Create and continue editing'.

Once you are done adding a document per quote, you should see the list back in the collection page:

The screenshot shows the 'Collection: quotes' page. The 'Documents' tab is selected. There are three documents listed:

- Document 1: _id: 575084aadcba0f329529d6ad, text: "The best preparation for tomorrow is doing your best today", author: "H. Jackson Brown". Actions: edit, delete.
- Document 2: _id: 575084cdcb0f329529d6b6, text: "If opportunity doesn't knock, build a door", author: "Milton Berle". Actions: edit, delete.
- Document 3: _id: 575084e2dcba0f329529d6bd, text: "Try to be a rainbow in someone's cloud", author: "Maya Angelou". Actions: edit, delete.

On the right, there is a sidebar titled 'Documents (aka Objects)' with information about the collection and a note about supported queries.

Let's now test the heroku local command:

```
~/graphql-project-test $ heroku local

[OKAY] Loaded ENV .env File as KEY=VALUE Format
[WARN] ENOENT: no such file or directory, open 'Procfile'
[OKAY] package.json file found - trying 'npm start'
11:59:08 AM web.1 | > NODE_ENV=production node server.js
11:59:09 AM web.1 | Connected to MongoDB server
11:59:09 AM web.1 | Running Express.js on port 5000
```

It looks like the CLI used port 5000 for this local run; we can test our application by navigating to <http://localhost:5000/>. We should see the quotes documents we added in our mlab MongoDB UI.

The heroku local command used the Node.js local installation that we had on the local machine. Heroku will not install the exact Node.js version we used locally unless we tell it to. We can do that using the engines section in package.json:

```
"engines": {
  "node": "6.3.1"
}
```

This will instruct Heroku to use Node.js version 6.3.1 and not its default one.

We're now ready to deploy to the cloud. All we need to do is commit the changes we made so far (ignoring the local .env file), and push them to the Heroku remote:

```
~/graphql-project-test $ echo .env >> .gitignore

~/graphql-project-test $ git add .gitignore package.json server.js

~/graphql-project-test $ git commit -m "Prepare for Heroku"
[prod 0f2242e] Prepare for Heroku
 3 files changed, 8 insertions(+), 4 deletions(-)
```

To deploy, we use git push. We do, however, need to push the local prod branch to the remote master branch. We can use this command:

```
~/graphql-project-test $ git push heroku prod:master

Counting objects: 18, done.
Delta compression using up to 8 threads.
Compressing objects: 100% (17/17), done.
Writing objects: 100% (18/18), 5.10 KiB | 0 bytes/s, done.
Total 18 (delta 0), reused 0 (delta 0)
remote: Compressing source files... done.
```

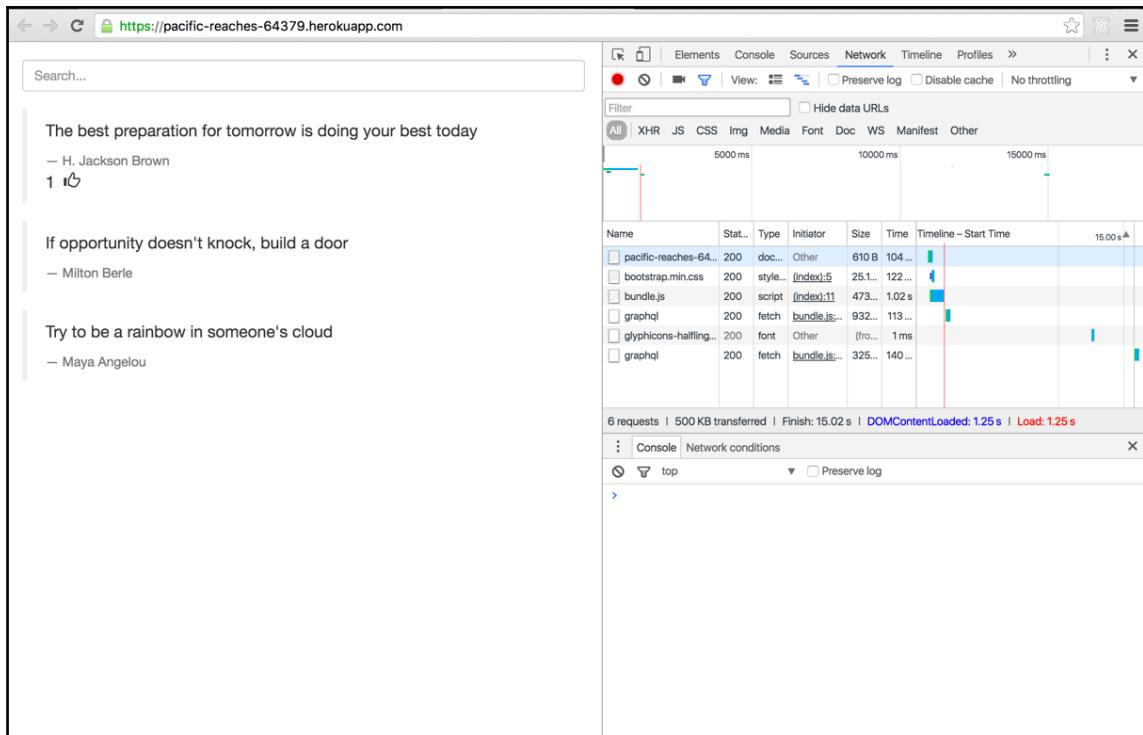
```
remote: Building source:
remote:
remote: ----> Node.js app detected
remote:
remote: ----> Creating runtime environment
...
remote:      Done: 17.8M
remote: ----> Launching...
remote:      Released v4
remote:      https://pacific-reaches-64379.herokuapp.com/ deployed to
Heroku
remote:
remote: Verifying deploy... done.
To https://git.heroku.com/pacific-reaches-64379.git
 * [new branch]      master -> master
```

We just handed our code to the Heroku platform and the platform will take it from there. It will install all the dependencies, generate our deployment files, run a Node server for our application, and make it available under a web server hosted at a domain matching our Heroku application name.

To launch a browser on that domain, we can use the following command:

```
~/graphql-project-test $ heroku open
```

This will launch the application in your browser, and you should see the quotes we added to the mlab MongoDB UI:



If you want to compare your code with what I deployed to my Heroku project, I pushed all the changes we made in the local `prod` branch to a deployed branch under my GitHub remote at <https://github.com/edgecoders/graphql-project>.

Summary

In this chapter, we looked at two different ways to deploy our GraphQL server and Relay application to the cloud. We first looked at an Infrastructure as a Service option, EC2 from AWS, and then looked at a Platform as a Service option, Heroku. For both options, we had to modify our project a little bit to meet certain requirements on these services. We used GitHub to share the code base with the services, and we changed our build process to bundle a production-optimized version of our application and its dependencies.

Index

A

- Amazon Web Services (AWS)
 - reference link 178
- Application Programming Interface (API)
 - tasks 10

C

- Cache layer 20
- connection model, Relay
 - about 21, 122, 123, 124, 126, 131
 - after/first model 21
 - offset/limit model 21
- core principles, Relay
 - connection model 21
 - object identification 20
 - storage and caching 20

D

- description property 59

E

- EC2 deployment
 - MongoDB, installing 184
 - Node.js server, running 185, 188
- EC2
 - deploying on 178, 179, 181, 183

G

- GitHub repository
 - reference link 64, 91
- GraphiQL 18
- GraphQL API
 - reference link 17
- GraphQL JavaScript
 - reference link 22

- graphql library
 - download link 24
- GraphQL query language syntax
 - aliases 49
 - directives 47
 - documents and operations 42
 - fields 44
 - fragments 51
 - mutations 54
 - variables 45
- GraphQL schema
 - about 55
 - example 89
 - introspection 59
 - resolve function 73
 - schema object 55
 - type schema 65
 - validation rules 85
 - versioning 86
- GraphQL server, setting up
 - about 22
 - field arguments, using 31
 - GraphiQL editor 39
 - HTTP interface, setting up 37
 - MongoDB, setting up 32
 - Node.js, installing 22
 - schema, defining 24
 - schema, using 28
 - two-dice roll, simulating 29
- GraphQL
 - about 6
 - defining, as language 6
 - defining, as runtime 7
 - features 10
 - used, for practical needs 12

J

JavaScript Object Notation (JSON)
reference link 7

L

like action implementation
about 140, 142, 143, 145
global ID, using 146, 148
Node interface 148, 149, 150, 151, 152
lodash library
reference link 136

M

MongoDB Node.js driver
installation link 34
MongoDB
installation link 32
mutation
invoking, Relay used 160, 162, 164, 165

N

Node Version Manager (NVM) 22
Node.js
installation link 22

O

optimistic updates 166, 167, 169

Q

Queue Store 20
quotes library 92
root-level field 113, 114, 116, 117

R

React applications
GraphQL, using without Relay 98
react-devtools
reference link 103
Read, Eval, Print, Loop (REPL) 23
Relay containers 101
Relay Node application deployment
GitHub repository, creating 171, 172
local deployment 173, 174, 177

preparing 170
Relay Store 20
Relay-compliantGraphQL mutation
defining 154, 156, 157, 158, 159
Relay
about 6, 8, 105
connection model 122
core principles 20
defining, as framework 8
features 19
first query operation, performing 117, 118, 119, 120, 122
GraphQL queries, transforming 105, 106, 108, 111, 112
optimistic updates 166
used, for building data-driven applications 9
request
validating 85
resolve function
about 73
args argument 74
context argument 78
info argument 79
resolving, with promises 80
source argument 74
RESTful APIs
reference link 15
versus GraphQL APIs 11

S

schema object 55
search feature, adding in Relay
about 135
Relay variables, using 136, 137, 138, 139
search form component, adding 135
search feature
adding, to GraphQL API 133, 134
implementing 132, 133
implementing, in Relay 135

T

t2.micro 179
type system
about 65
ENUM type 72

interface and unions 66
scalars and object types 66
type modifiers 70

V

versioning 86

W

Webpack
setting up 94