

# Get Going with GraphQL

Learn How to Build JavaScript Applications  
with Apollo Server and Apollo Client

Mandi Wise

# **Get Going with GraphQL**

**Learn How to Build JavaScript Applications with Apollo Server and Apollo Client**

**Mandi Wise**



**8-bit press**

## **Get Going with GraphQL**

By Mandi Wise

Copyright © 2021 8-Bit Press Inc. All rights reserved.

Published by 8-Bit Press Inc.

<https://8bit.press>

No part of this publication may be reproduced, stored in a retrieval system, or transmitted, in any form by means electronic, mechanical, photocopying, or otherwise without out prior written permission of the publisher, except for brief quotations embodied in articles or reviews. Thank you for respecting the hard work of the author.

While the advice and information in this book is believed to be true and accurate at the date of publication, the publisher and the author assume no legal responsibility for errors or omissions, or for damages resulting from the use of the information contained herein.

First Edition, July 2021

**Illustrator:** Mandi Wise

Cover by Mandi Wise

ISBN 978-1-63901-691-4

*For Maxwell*



# Contents

<b>Preface</b>	<b>ix</b>
What's Inside . . . . .	x
Who Should Read this Book . . . . .	x
Getting the Most from this Book . . . . .	xi
Formatting Conventions . . . . .	xi
Code Blocks . . . . .	xi
Inline Code . . . . .	xii
Info Boxes . . . . .	xii
Quotes . . . . .	xii
Package Versions . . . . .	xii
Reference Source Code. . . . .	xii
Required Software . . . . .	xiii
Optional Software . . . . .	xiii
The Game Plan . . . . .	xiii
<b>Chapter 1   Up and Running with GraphQL and Apollo Server</b>	<b>1</b>
Hello Schema, Nice to Meet You . . . . .	1
Basic Types of Types . . . . .	4
Scalar Types. . . . .	4
Object Types . . . . .	5
Two Ways to Modify Types. . . . .	6
Non-Null . . . . .	6
Lists . . . . .	7
A GraphQL API to Call Our Own . . . . .	7
But First, a (Mocked) REST API . . . . .	9
Initial Type Definitions . . . . .	11
Resolvers, aka Functions that Do the Data Fetching. . . . .	13
Wire Up Apollo Server . . . . .	17
Exploring the GraphQL API. . . . .	18
Summary. . . . .	23

<b>Chapter 2   More on Queries</b>	<b>24</b>
Best Practice: Design the Schema with Clients in Mind . . . . .	24
Organize Code with an Apollo Data Source . . . . .	27
Add a Review Type Definition . . . . .	32
Add a User Type Definition . . . . .	38
Best Practice: Use Operation Names and Variables . . . . .	46
DRYer Operations with Fragments . . . . .	48
Summary . . . . .	49
<b>Chapter 3   Mutating Data</b>	<b>51</b>
Mutations: How We Write Data via a GraphQL API . . . . .	51
Create an Author . . . . .	52
Create a Book with an Input Type . . . . .	54
Create a Review with Custom Error Handling . . . . .	58
Update or Delete a Review . . . . .	63
Sign Up a New User . . . . .	66
Update a User's Library . . . . .	70
Summary . . . . .	74
<b>Chapter 4   Pagination and Search Queries</b>	<b>75</b>
Book Genres as an Enum . . . . .	75
Pagination Options for GraphQL APIs . . . . .	79
Offset-Based . . . . .	79
Cursor-Based . . . . .	80
Relay-Style . . . . .	81
So Which Do We Choose? . . . . .	82
Add Pagination for Authors, Books, Reviews, and Users . . . . .	82
Interfaces and Unions: What Are They Good For? . . . . .	98
Add a Query to Search for People by Name . . . . .	100
Add a Query to Search Books by Author or Title . . . . .	105
Summary . . . . .	109
<b>Chapter 5   Documentation, Custom Scalars, and Custom Directives</b>	<b>110</b>
Best Practice: Document the Schema . . . . .	110
Add a Custom DateTime Scalar . . . . .	121
Add a Rating Scalar . . . . .	124
Directives in GraphQL . . . . .	128
Add an @unique Custom Directive . . . . .	129
Summary . . . . .	136
<b>Chapter 6   Authentication and Authorization</b>	<b>137</b>
Locking Down a GraphQL API . . . . .	137

What's a JSON Web Token? . . . . .	138
Save a Hashed Password on User Sign-up . . . . .	141
Add a login Mutation . . . . .	147
Migrate to Apollo Server Express . . . . .	150
Set Valid, Decoded JWTs on Apollo Server's Context . . . . .	153
Add a viewer Query . . . . .	155
Add GraphQL Shield for Authorization . . . . .	158
Summary . . . . .	164
<b>Chapter 7   React App Set-up</b>	<b>165</b>
Create a React App . . . . .	165
Add Tailwind CSS . . . . .	169
Install React Router . . . . .	171
Style the Main Layout . . . . .	174
Style the Login Page Layout . . . . .	179
Summary . . . . .	187
<b>Chapter 8   Apollo Client with User Authentication</b>	<b>188</b>
Install Apollo Client . . . . .	188
Add Books to the Index Page . . . . .	190
Send the JWT in a Cookie . . . . .	196
Log In or Register a New User from the Client . . . . .	204
Use Context and Hooks to Manage Authentication State . . . . .	212
Log Out a User . . . . .	218
Render Private and Public Routes . . . . .	221
Display a User's Library on the Home Page . . . . .	224
Summary . . . . .	229
<b>Chapter 9   Pagination, Mutations, and the Apollo Client Cache</b>	<b>230</b>
Paginate the List of Books on the Index Page . . . . .	230
Display a Single Book with Details . . . . .	237
Add and Remove Books from a Library . . . . .	246
Add Pagination to the Home Page . . . . .	254
Add Book Search . . . . .	256
Create a Book Review . . . . .	262
Update a Book Review . . . . .	274
Delete a Book Review . . . . .	281
Create a New Book with Authors . . . . .	285
Summary . . . . .	296
<b>Chapter 10   Real-time Updates with Subscriptions</b>	<b>298</b>
Support a Subscription for New Reviews . . . . .	298

Connect to the WebSocket Endpoint with Apollo Link . . . . .	304
Optional: Add an Apollo Link to Handle Expired JWTs . . . . .	308
Add a Subscription Operation to the Client . . . . .	311
Summary . . . . .	314
<b>Appendix A: GraphQL Resources</b>	<b>316</b>
<b>About the Author</b>	<b>319</b>
<b>Changelog</b>	<b>320</b>

# Preface

Welcome to *Get Going with GraphQL!* I'm thrilled you've chosen this book to dive into learning about full-stack application development with GraphQL APIs.

GraphQL was invented at Facebook in 2012 and was publicly launched in 2015. Since then, it has opened up a world of new possibilities for API design and consumption. Today, it has been famously adopted by companies including The New York Times, Shopify, Coursera, AirBnB, GitHub, and Netflix. It has also inspired countless open source projects related to it, including the popular suite of Apollo tools for working with GraphQL APIs in both client and server applications.

It may seem like everyone is talking about these days, but if you're new to GraphQL you may still be wondering what all of the fuss is about. If you visit the [graphql.org](https://graphql.org) homepage, you will see GraphQL described as "a query language for your API." That's pretty high-level, so from a more tactical perspective, the GraphQL specification outlines the mechanics of that query language as well as the runtime for responding to query requests received by an API. And despite what the "QL" in its name might suggest, it has nothing to do with any particular database implementation.

GraphQL's appeal stems largely from its orientation toward making API consumption easier for clients. Unlike a REST API, a GraphQL API is exposed through a single endpoint and will often stand between a client and any number of backing data sources. A well-designed GraphQL schema can also help client developers avoid the common REST-related headache of simultaneously over-fetching and under-fetching the data needed to render a user interface. What's more, GraphQL puts client developers in the driver's seat to query only the data that they need and in the shape that they need it. And with the right schema governance practices and observability tools in place, a GraphQL API can be safely evolved without versioning. That means that a GraphQL schema can be iterated to support new product requirements as they emerge and increase the velocity at which new features are released to users.

With all of this in mind, it's easy to see how GraphQL knowledge is a must-have for JavaScript developers today and this book has been thoughtfully designed to help you ramp up quickly. Throughout this book, we'll explore all of GraphQL's essential features, as well as how Apollo Server and Apollo Client may be used to supercharge API and client application development.

## What's Inside

This book covers a wide range of topics relevant to developing web applications from scratch with JavaScript and does so using a practical, project-based approach. Working through this book from start to finish will leave you with a relatively full-featured React application powered by a GraphQL API.

Specific topics covered throughout the book include:

- Setting up Apollo Server
- Working with built-in Scalar types
- Defining Object, Input Object, Enum, Interface, and Union types in a schema
- Supporting queries, mutations, and subscriptions with a GraphQL API
- Employing best practices such as using operation names and query variables
- Using an Apollo data source to fetch data in resolvers
- Paginating results received from a GraphQL query
- Creating custom scalars and directives
- Adding documentation to a GraphQL API
- Adding basic authentication to a Node.js application using Express middleware and JSON Web Tokens (JWTs)
- Adding authorization to a GraphQL API on a field-by-field basis
- Setting up a React application with Apollo Client 3
- Sending query, mutation, and subscription operations to a React application
- Sending authenticated requests from Apollo Client to a GraphQL API
- Leveraging Apollo Client's normalized cache to reduce network requests

In its exploration of these topics, this book is divided into 10 chapters. Chapters 1 to 6 focus entirely on the server side concerns of designing and building a GraphQL API with Apollo Server. Chapters 7 to 9 shift gears to the client side and cover the development of a React web application that consumes the GraphQL API. Chapter 10 wraps up the book by configuring subscription operation support on both the server and client applications.

## Who Should Read this Book

This book assumes you have little or no prior knowledge of GraphQL, but does assume you have some experience building React applications. Before proceeding, consider if you have at least:

- Intermediate knowledge of JavaScript (including ES2015+ features of the language)
- Some experience building front-end user interfaces for the web with React (no experience with hooks is required)
- Some knowledge of how a REST API works (because we will use one as a backing data source for the GraphQL API)
- Experimented with Node.js in the context of web application development

If those qualifications sound like they describe you, then you're in the right place. By the time you've finished this book, you'll feel confident drawing on many of the features of Apollo Server and Client and applying them to real-world development scenarios.

## Getting the Most from this Book

To get the most from your experience reading this book, I encourage you to work through the chapters chronologically because the code in each chapter builds on the previous one. This book and its chapters are structured around building out a single full-stack application, beginning with a GraphQL server and ending with a React application that consumes the API.

If you intend to build out the demonstration application as you work through the chapters, I also encourage you to explicitly type out as much of the code that you encounter as possible rather than simply copying and pasting snippets. In my experience, taking the time to intentionally type out code examples aids comprehension and builds mental muscle memory.

Of course, if you're working on another GraphQL-based project right now and are in search of insights or code examples for any of the aforementioned topics, each chapter of this book also stands alone in building out a complete, logical chunk of functionality for the application. You will almost certainly find that various chapters serve as helpful reference points for that work too.

## Formatting Conventions

### Code Blocks

In favor of brevity, some code examples are truncated where code that was previously added to a file doesn't change in the context of a new update to that file. Additionally, file diffs are highlighted in code examples to enhance readability and filenames are added above the code blocks in italics, where applicable. For example:

*server/src/index.js*

```
const typeDefs = gql`  
  # ...  
  
  type Query {  
    author(id: ID!): Author  
    authors: [Author]  
    book(id: ID!): Book  
    books: [Book]  
  }  
`;
```

## Inline Code

Code and filenames that are referenced inline (for example, within a paragraph or list item) are rendered in this monospaced font to distinguish them from the other text.

## Info Boxes

Supplementary notes and additional tips related to the main content are presented in gray boxes throughout the chapters as follows:

You can find a current version of the GraphQL specification here: <http://spec.graphql.org/>

## Quotes

Longer sections of text directly quoted from another source are delineated with a gray border to the left of the quote:

The best programs are written so that computing machines can perform them quickly and so that human beings can understand them clearly. A programmer is ideally an essayist who works with traditional aesthetic and literary forms as well as mathematical concepts, to communicate the way that an algorithm works and to convince a reader that the results will be correct.

—Donald E. Knuth, *Selected Papers on Computer Science*

## Package Versions

For compatibility purposes, package versions are specified wherever you are instructed to install a package from npm. You may wish to experiment with updated package versions as they become available, but compatibility with the other code in this book cannot be guaranteed.

## Reference Source Code

The complete source code for the application built throughout this book may be accessed at:

<https://github.com/8bitpress/get-going-with-graphql-source-code>

## Required Software

To build the application as outlined in the chapters that follow you will need to have **Node.js 14.13.0 or higher** installed on your computer. This version is required because it was the first version of Node.js that provided support for importing named exports from CommonJS modules using ES module syntax. You can download the latest LTS version of Node.js for your operating system here:

<https://nodejs.org/en/download/>

## Optional Software

Optionally, if you use VS Code as an editor, then also consider installing the **Apollo GraphQL extension** to enhance your development experience with features such as GraphQL syntax highlighting:

<https://marketplace.visualstudio.com/items?itemName=apollographql.vscode-apollo>

And to facilitate client-side development, you may also wish to install the Apollo Client Developer Tools for Chrome or Firefox:

<https://www.apollographql.com/docs/react/development-testing/developer-tooling/#apollo-client-devtools>

## The Game Plan

In the chapters that follow, we'll build out a GraphQL API to be consumed by React application called *Bibliotech*. The client application will allow unauthenticated users to view details and reviews about books contained within the Bibliotech catalog. Users can also sign up for an account and create libraries of their favorite books and also submit and update reviews of their own when logged in.

We have a lot of code to write to get from zero to full-stack application at the end of this book, so fire up your code editor and let's go!

## Chapter 1

# Up and Running with GraphQL and Apollo Server

In this chapter, we will:

- Try running a query against the GitHub GraphQL API
- Use built-in Scalar types, define Object types, and create non-null and list fields
- Set up an Apollo Server with some basic type definitions and corresponding resolvers
- Add arguments to fields
- Run queries against a locally-running GraphQL API using Apollo Studio Explorer

## Hello Schema, Nice to Meet You

The heart of any GraphQL is its schema. While a REST API provides a series of structured routes that can be used to read or write different resources based HTTP verbs, a GraphQL API will instead expose a single endpoint (often at `/graphql`) to query any of the data exposed by the API. If you've spent many years working with REST APIs, then you may immediately wonder how a single endpoint can do all of the heavy lifting for an API.

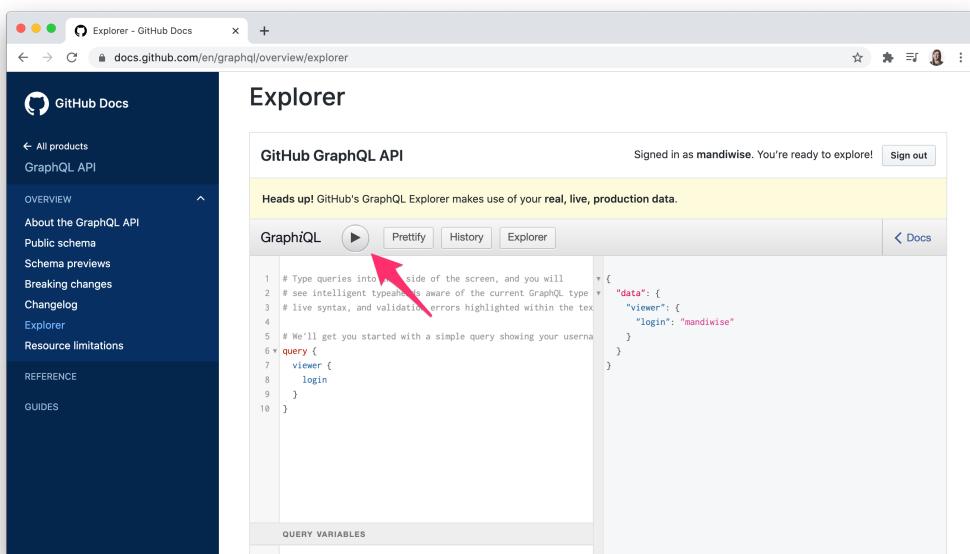
The answer is in the specially formatted query that we send to the GraphQL API endpoint. When sending requests to the API via HTTP, this query is often included in the body of a POST request. The GraphQL query language allows us to describe exactly what data we want to receive and in what shape we'd like to receive and—just like that—the GraphQL API will send a response back in precisely the format we requested as long as it's compliant with API's schema. If you've ever felt the pain of simultaneously over-fetching some data from an API while under-fetching the data you really wanted, and then subsequently needing to juggle various asynchronous requests and mash their responses together to finally populate all of the required data for a single view, then GraphQL will feel like a breath of fresh air for you.

## Convention versus Specification

The [GraphQL specification](#) is a detailed document describing how to define a GraphQL schema, how to write queries against that schema, and how to build an execution engine (also known as a GraphQL server) to respond to those queries. It does not, however, specify anything about the transport mechanism used to shuttle requests to and from a GraphQL server. That said, HTTP is commonly chosen for the transport layer and GraphQL clients often send POST requests containing a serialized JSON representation of the query in the request body. Similarly, GraphQL servers often respond with JSON as well.

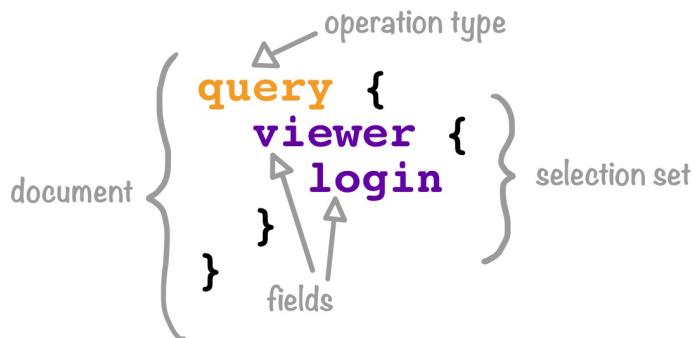
The POST verb is a common default for GraphQL requests because GET requests have size limits and complex queries may grow quite large. Additionally, a POST request's body is encrypted over HTTPS, which will help limit potential exposure of any sensitive operation argument values included with the request. That said, tools like [Automatic Persisted Queries](#) allow you to hash GraphQL queries to ensure that they fit comfortably within GET requests limits, and thus allow you to take advantage of browser caches and CDNs.

GraphQL's query language is very intuitive and the best way to learn it is to start writing queries. If you have a GitHub account, you can try sending a query to the GitHub GraphQL API using a browser-based tool available at <https://docs.github.com/en/graphql/overview/explorer>. After logging in with your GitHub account, the [GraphQL IDE](#) will be active with a `viewer` query preloaded in it. Press the play button above the editor to view the response:



With an easy click of a button, you just made your first query to a GraphQL API! And as promised, you were able to query the API for a specific piece of information about the currently logged in-user (in this case, your username), and that data was returned as JSON and in a shape mirroring the original query.

Now that we've seen a query in action, let's break it down piece by piece:



The entirety of what you see above is known as a *document* in GraphQL speak and that document contains an *operation*. In this case, it is a *query* operation that performs a read-only fetch operation, but there are also *mutation* operations that write and fetch data and there are *subscription* operations that can fetch data in real-time in response to source events.<sup>1</sup>

When performing query operations only we can optionally omit the *query* keyword in front of the curly brackets as a shorthand:

#### *GraphQL Query*

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

Inside the outermost curly brackets, we are then left with the *selection set* of *fields* that specify the precise data that we want to receive in the response. Lastly, note that GraphQL syntax also supports comments by preceding comment text with the # character.

We've just unpacked several important GraphQL features using this basic query, but some important questions still linger. For one, how would we know that we were able to query a *viewer* from the GitHub API in the first place? And how do we find out what other fields can be queried for a *viewer*?

<sup>1</sup><https://spec.graphql.org/June2018/#sec-Language>

The answer to those questions is the schema—a GraphQL schema defines what *types* and *fields* are queryable through the API, as well as the relationships between types. A GraphQL schema also defines the *root operation types* that it supports, including `query` (required), `mutation` (optional), and `subscription` (optional). We can think of the fields belonging to these root operation types as the entry points to a GraphQL API.

One popular way to define a GraphQL schema is with the *schema definition language (SDL)*. For example, to support queries for a `viewer` field on the `query` root operation type, the GitHub API contains the following definition:

```
type Query {  
  viewer: User!  
}
```

Above, the `query` root operation type is defined as an *Object* type called `Query` and that type contains a single `viewer` field (in reality, the GitHub API’s `Query` type contains many more fields, but we’ll start with one to keep things simple). But where does `User` come from, and what is its relation to the nested `login` field we saw earlier? And what does the exclamation point after `User` mean? To answer these questions, we’ll need to dive into GraphQL’s *type system*.

## Basic Types of Types

According to the GraphQL specification, types are the “fundamental unit” of a schema.<sup>2</sup> We just saw an example of an Object type when we defined type `Query` with its `viewer` field above, and there are five other *named* types available in GraphQL that we can use too.

### Scalar Types

The most basic type in GraphQL is a *Scalar* type. We can define custom Scalar types, but we typically start with the ones built into GraphQL. Out of the box, we can use `Int`, `Float`, `String`, `Boolean`, and `ID` Scalar types in a schema. The first four of those Scalar types represent exactly what they suggest. The `ID` type is a special-purpose Scalar type to indicate that a field is a unique identifier and that it will typically not be human-readable. Even if the `ID` value is numeric it should always serialize to a string.

In practice, the built-in Scalar types are used to specify the types that correspond to fields in a schema as follows:

```
type User {  
  bio: String  
  id: ID
```

---

<sup>2</sup><https://spec.graphql.org/June2018/#sec-Type-System>

```
    isViewer: Boolean  
    pinnedItemsRemaining: Int  
}
```

## Object Types

But wait! We haven't addressed how we were able to arbitrarily define type `User` in the last example. We have already seen an example of an Object type when we defined type `Query` previously, but we can also define Object types that are specific to our schema.

Object types are very powerful because they allow us to express relationships (or *edges*) between the different data types (or *nodes*) within our graph. If we think of the `query` root operation type as an entry point to our API, then Scalar types are the outer nodes in our graph where we finally reach the data. By connecting Object types through their fields, we can facilitate deeper traversal across the graph to these outer nodes.

In our simplified GitHub API schema, we can move from the root `Query` type to a single authenticated `User` via the `viewer` field. With the following modification, we can go one level deeper:

```
type User {  
  bio: String  
  id: ID  
  isViewer: Boolean  
  pinnedItemsRemaining: Int  
  status: UserStatus  
}  
  
type UserStatus {  
  emoji: String  
  id: ID  
  message: String  
}
```

Here we see a powerful feature of GraphQL's type system in action—the ability to define Object types as collections of related fields and then nest them within one another to define the relationships between those types that ultimately allow us to move from node to node across our data graph.

Much of a GraphQL schema will end up being composed of the Object types that you create, but there are still two specification-defined Object types that we should make note of before moving on. Those types are the `Mutation` and `Subscription` types that correspond to those root operations in an API. We'll cover those types in-depth in Chapter 3 and Chapter 10 respectively.

And if you’re keeping count, you’ve probably noticed that we’ve only explored two of the six named types in GraphQL so far. Scalar and Object types will be all we need to begin building a functioning GraphQL API, so we’ll cover *Input* types in Chapter 3 and *Enum*, *Interface*, and *Union* types in Chapter 4 as we require them to build out additional API features.

## Two Ways to Modify Types

By default, GraphQL expects that a field’s corresponding type is singular and that it may return null values. However, in addition to the named types we’ve already explored, the GraphQL specification outlines two different *wrapping types* that we can use “wrap” named types and modify their defaults.

### Non-Null

To make a field non-null, we use an exclamation point. Making a field non-null means that a client consuming this API can always expect a value to be returned for this field, and if one is not available, then the query will return an error. We have already seen this where we defined the `viewer` field on the `Query` type:

```
type Query {  
  viewer: User!  
}
```

As a further example, if we update the `User` type to reflect GitHub’s real API, then we would make the `id`, `isViewer`, and `pinnedItemsRemaining` fields non-null:

```
type User {  
  bio: String  
  id: ID!  
  isViewer: Boolean!  
  pinnedItemsRemaining: Int!  
  status: UserStatus  
}
```

### Best Practice: Think About Nullability Up Front

Non-null fields have the advantage of making API responses more predictable, but they come with the trade-off of making that API harder to evolve in some ways in the future. For example, if a non-null field is later transitioned to nullable or removed entirely from an Object type, then breaking changes may result for clients that assume the field value will be there.

The inherently evolvable nature of a GraphQL API combined with the risk of breaking changes where alterations to a field's nullability are concerned is a good reason to take time to make an informed choice about whether a field should be nullable or non-null when it's added to a schema.

When unforeseen circumstances require changes to field nullability in the future, you can use observability tooling to understand what clients are using that field in your API and then proactively help those affected client developers avoid breaking changes to user interfaces.

## Lists

Fields can also output lists of types and we indicate such a list by wrapping the type name in brackets. If we wanted to add a non-null field to the `User` type that contained a list without any null items, we would update our code as follows:

```
type User {  
  bio: String  
  id: ID!  
  isViewer: Boolean!  
  pinnedItemsRemaining: Int!  
  status: UserStatus  
  organizationVerifiedDomainEmails: [String!]!  
}
```

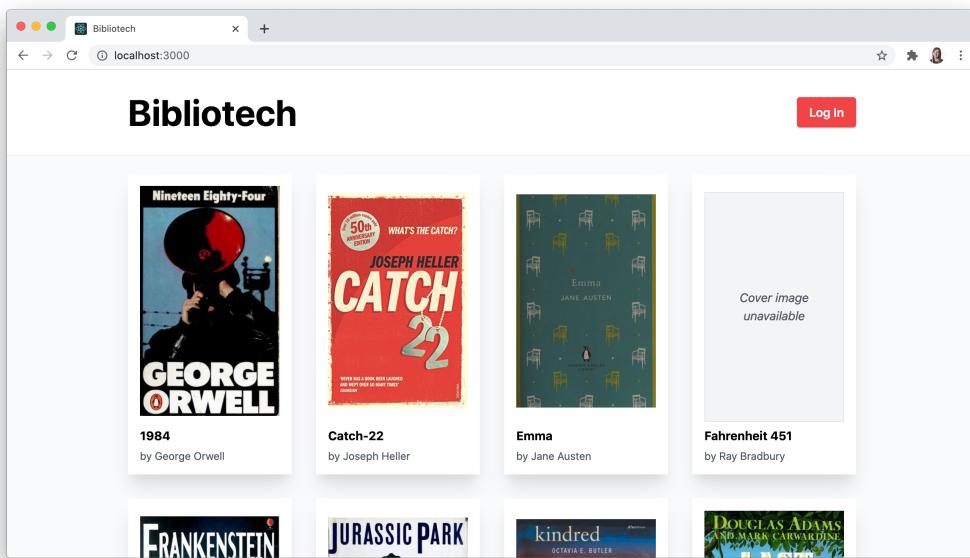
If we wanted to permit a potentially empty list or null items in the list, but still require an empty list to be returned (rather than a null value), then we would specify the field's output type as `[String]!`. Similarly, if we wanted to allow the field itself to be nullable but disallow non-null values in the list, we would specify the field's output type as `[String!]`. Lastly, to permit a nullable list of strings that may contain null values or be empty, we would specify the field's output type as `[String]`.

## A GraphQL API to Call Our Own

We have now covered just enough GraphQL theory to roll up our sleeves and start building a GraphQL API from scratch with Apollo Server. To do that, we'll need to define a schema and set up a GraphQL server to respond to client requests to our API.

Our objective is to build a new and improved version of the REST-based Bibliotech API using GraphQL. Under the hood, the new GraphQL API will use the existing REST endpoints as a data source—a step that many teams take to migrate to GraphQL within an organization. Later on, as the REST API is phased out, different data sources may be swapped in behind the GraphQL API and any clients presently making GraphQL requests will be none the wiser.

The Bibliotech API provides information about books, authors, and allows users to create a personal library of their favorite books, as well as rate and review them. In the final three chapters, we'll also build out an MVP client application using React and Apollo Client to consume data from the new GraphQL API. The index page of the Bibliotech React app will look like this:



We'll leverage the GraphQL API to build out additional pages to display a user's favorite books, book search results, and a single book view. We will also need to create forms to authenticate users, add and update reviews, and add new authors and books. To begin, let's create a project directory, cd into it, and then run the following command to set up the server directory:

```
mkdir server && cd server
```

Second, we'll create a `package.json` file in the new `server` directory (the `--yes` flag creates the `package.json` file without asking any questions):

```
npm init --yes
```

Next, we'll install some initial dependencies:

```
npm i apollo-server@2.22.2 concurrently@5.3.0 dotenv@8.2.0 graphql@15.5.0  
      json-server@0.16.3 node-fetch@2.6.1 nodemon@2.0.7
```

Let's take a brief look at each of the packages we just installed:

- **apollo-server:** Apollo Server is an open-source GraphQL server that will allow us to define a GraphQL schema and execute queries against that schema.
- **concurrently:** We can run multiple commands at the same time using this package. It also has support for shortened commands with wildcards.
- **dotenv:** This package loads environment variables from a `.env` file into `process.env`.
- **graphql:** Apollo Server requires this library as a peer dependency.
- **json-server:** This package allows us to instantly spin up a REST API using data from a JSON file, so we'll use this to mock the existing Bibliotech API. You may wish to quickly review the [JSON Server documentation](#) if you haven't used it before.
- **node-fetch:** This is a Node.js implementation of `window.fetch` and it will facilitate fetching data from the REST endpoints.
- **nodemon:** Nodemon will automatically reload our server when files change in the project directory.

As an additional piece of configuration in the package `.json` file, we'll set the `type` field to `module` so that we can use [ECMAScript module syntax](#) in our files:

`server/package.json`

```
{  
  // ...  
  "type": "module",  
  // ...  
}
```

## But First, a (Mocked) REST API

Before we can use the Bibliotech REST API as a backing data source for the GraphQL API, we'll need to create a file from which JSON Server can query data. Create a `db.json` file in the `server` directory and add the following code to it:

`server/db.json`

```
{  
  "authors": [  
    {  
      "id": 1,  
      "name": "Douglas Adams"  
    }]
```

```
  ],
  "books": [
    {
      "id": 1,
      "cover": "http://covers.openlibrary.org/b/isbn/9780671461492-L.jpg",
      "summary": "After a Vogon constructor fleet destroys Earth to make way for a hyperspace bypass, the last surviving man, Arthur Dent, embarks on an interstellar adventure with his friend Ford Prefect (who, apparently, was an alien all along).",
      "title": "The Hitchhiker's Guide to the Galaxy"
    }
  ],
  "bookAuthors": [
    {
      "id": 1,
      "authorId": 1,
      "bookId": 1
    }
  ]
}
```

If you're using version control in this project, you may wish to ignore the `db.json` file as we will be writing test data to this file as we build out the GraphQL API throughout this book.

JSON Server will automatically create an endpoint for each of the top-level properties in the `db.json` file. We'll primarily use the `/authors` and `/books` endpoints for now, while the `/bookAuthors` endpoint will act like a join table would to capture the many-to-many relationship between authors and books. Next, we can make our lives easier by writing some custom routes for the REST API that will make the process of querying for all authors of a given book or all books by a given author a bit more intuitive than querying `/bookAuthors` directly. To do this, we'll create a `routes.json` file in the `server` directory and add the following code to it:

`server/routes.json`

```
{
  "/authors/:authorId/books":
    "/bookAuthors?authorId=:authorId&_expand=book",
  "/books/:bookId/authors": "/bookAuthors?bookId=:bookId&_expand=author"
}
```

Lastly, we can remove the existing `test` script in the `package.json` file and replace it with a script to start up our brand new REST API:

`server/package.json`

```
{  
  // ...  
  "scripts": {  
    "server:rest": "json-server -w db.json -p 5000 -r routes.json -q"  
  },  
  // ...  
}
```

Run `npm run server:rest` and try out the following endpoints to confirm they return the expected data before proceeding:

- `http://localhost:5000/authors`
- `http://localhost:5000/books`
- `http://localhost:5000/authors/1/books`
- `http://localhost:5000/books/1/authors`

## Initial Type Definitions

With our REST API up and running, it's time to define the first few types in the Bibliotech schema. First, we'll create a subdirectory to organize our GraphQL-related code, as well as a `typeDefs.js` file to house our type definitions. We'll create a `graphql` directory in `server`, add the `typeDefs.js` file to it, and then set up the new file as follows:

`server/src/graphql/typeDefs.js`

```
import { gql } from "apollo-server";  
  
const typeDefs = gql`  
  # GraphQL type definitions will go here...  
`;  
  
export default typeDefs;
```

We're going to define our type definitions as a string inside of a JavaScript file using SDL and then wrap that string in the `gql` template literal tag to convert it into the format that Apollo Server is expecting. As a bonus, the `gql` tag will enable GraphQL syntax highlighting within the string when the [Apollo GraphQL VS Code extension](#) is installed.

Next, let's add some initial type definitions for the `Author` and `Book` Object types, as well as the root `Query` type with some related fields for querying lists of authors and books:

`server/src/graphql/typeDefs.js`

```
import { gql } from "apollo-server";

const typeDefs = gql`  
  type Author {  
    id: ID!  
    books: [Book]  
    name: String!  
  }  
  
  type Book {  
    id: ID!  
    authors: [Author]  
    title: String!  
  }  
  
  type Query {  
    authors: [Author]  
    books: [Book]  
  }  
`;  
  
export default typeDefs;
```

Getting lists of authors and books is useful, but we'll also need a way to query a single author or book based on a unique identifier. To do that, we'll add some new query fields with *arguments*. In our previous example with `User` Object type from the GitHub API, we mocked the `organizationVerifiedDomainEmails` field on the type as it is in the real API, but left out one key detail. This field accepts a `login` argument that represents the login of the organization from which to match verified domains. The real field looks like this:

```
type User {  
  # ...  
  organizationVerifiedDomainEmails(login: String!): [String!]!  
}
```

Parentheses are used after the field name to indicate that the field accepts arguments, and inside, each of the arguments is listed by name with its corresponding type. In this above example, the `login` argument expects a non-null `String` to be provided when the field is queried.

For our API, we'll add `author` and `book` fields that each accept a non-null ID as an argument, and return a single Author or Book respectively:

`server/src/graphql/typeDefs.js`

```
import { gql } from "apollo-server";

const typeDefs = gql`  
# ...  
  
type Query {  
  author(id: ID!): Author  
  authors: [Author]  
  book(id: ID!): Book  
  books: [Book]  
}  
;  
  
export default typeDefs;
```

The basic syntax for a query that contains field arguments is as follows:

*GraphQL Query*

```
query {  
  author(id: "1") {  
    name  
  }  
}
```

## Resolvers, aka Functions that Do the Data Fetching

With some type definitions in place, we now need a way to respond to API requests with real data. To do that, we must write *resolver* functions. The Apollo Server documentation<sup>3</sup> provides the following concise definition of what a resolver function does:

A resolver is a function that's responsible for populating the data for a single field in your schema. It can populate that data in any way you define, such as by fetching data from a back-end database or a third-party API.

<sup>3</sup><https://www.apollographql.com/docs/apollo-server/data/resolvers/>

Resolvers have the following function signature:

```
someField(parent, args, context, info) {  
  return "Hello, world!";  
}
```

Let's explore what each of the resolver's parameters exposes in the function:

- **parent**: GraphQL resolvers are executed in a chain starting at the top-most level of the query and working down toward the Scalar types at the outer edges. At each step, the field currently being resolved will have access to the data from the previously resolved parent field in the chain. For the query, mutation, and subscription root operation types, this value will come from the `rootValue` function passed to the `ApolloServer` constructor. You'll often see this parameter named to reflect what the parent type is (for example, instead of naming the parameter `parent` it will be named `book` in field resolvers for the `Book` type).
- **args**: This parameter is an object that contains any arguments supplied for this field as a part of the query. For the example in the previous section, we would expect the `args` value of the `author` field resolver to be `{ id: "1" }`.
- **context**: The `context` parameter is another object and it provides us with a way to share data across all of the field resolvers for a given operation. For example, we may wish to include information from a decoded token to authorize access to different fields in our API. The `context` object is recreated with each request so we don't have to worry about the data contained within going stale or inadvertently being shared across requests.
- **info**: The final `info` parameter is typically needed for advanced use cases only and contains information about the field in question, a representation of the entire schema, an abstract syntax tree (AST) for the operation, and more.

### Good to Know!

Resolver functions can return promises, so it's OK to use the `async` keyword with them.

We'll keep our resolvers organized in a separate file, so create a `resolvers.js` file in the `server/src/graphql` directory. We'll structure the resolver functions in an object so that Apollo Server will be able to understand what resolvers correspond to which fields. We'll begin by adding the following code to `resolvers.js`:

`server/src/graphql/resolvers.js`

```
const resolvers = {  
  Query: {  
    async author(root, { id }, context, info) {
```

```
// Fetch an author by ID here...
},
async authors(root, args, context, info) {
  // Fetch all authors here...
},
async book(root, { id }, context, info) {
  // Fetch a book by ID here...
},
async books(root, args, context, info) {
  // Fetch all books here...
}
};

export default resolvers;
```

Note that each key nested in the object value of the `Query` key directly corresponds to a field name from the type definitions we just created. We have also marked each function as `async` because we must await a GET request to the mocked REST API to obtain the data that will be returned for the fields. Let's update each resolver to fetch the appropriate data now:

`server/src/graphql/resolvers.js`

```
import fetch from "node-fetch";

const baseURL = "http://localhost:5000";

const resolvers = {
  Query: {
    async author(root, { id }, context, info) {
      const res = await fetch(`#${baseURL}/authors/${id}`).catch(
        err => err.message === "404: Not Found" && null
      );
      return res.json();
    },
    async authors(root, args, context, info) {
      const res = await fetch(`#${baseURL}/authors`);
      return res.json();
    },
    async book(root, { id }, context, info) {
      const res = await fetch(`#${baseURL}/books/${id}`).catch(
        err => err.message === "404: Not Found" && null
      );
      return res.json();
    }
  }
};

export default resolvers;
```

```
    );
    return res.json();
},
async books(root, args, context, info) {
  const res = await fetch(` ${baseUrl}/books`);
  return res.json();
}
};

export default resolvers;
```

This code is all we need to take care of resolving data for fields on the `Query` type, but what about the `Author` and `Book` types? As you can imagine, if a schema contained many `Object` types with many different fields, then it could get very tedious to write resolvers for each field.

Luckily, if the parent object argument contains a property with a name that matches the field name, then Apollo Server will use a *default resolver* to automatically supply that value for the field. That means that `id` and `name` fields will be automatically resolved for the `Author` type and the `id` and `title` fields will be automatically resolved for the `Book` type because these properties are readily available in the objects fetched from the REST API.

However, because we expressed a many-to-many relationship between authors and books in our schema, we will need to provide a `books` field resolver for `Author` and an `authors` field resolver for `Book`. To do that, we'll update our code:

`server/src/graphql/resolvers.js`

```
// ...

const resolvers = {
  Author: {
    async books(author, args, context, info) {
      const res = await fetch(` ${baseUrl}/authors/${author.id}/books`);
      const items = await res.json();
      return items.map(item => item.book);
    }
  },
  Book: {
    async authors(book, args, context, info) {
      const res = await fetch(` ${baseUrl}/books/${book.id}/authors`);
      const items = await res.json();
      return items.map(item => item.author);
    }
  }
};

export default resolvers;
```

```
        },
    },
    Query: {
        // ...
    }
};

export default resolvers;
```

We now have all of the required code in place to resolve every field in the schema.

## Wire Up Apollo Server

Now that we have type definitions and resolvers ready to go, we're finally ready to create a new Apollo Server and start sending requests to the GraphQL API. We'll instantiate an `ApolloServer` object in a top-level `index.js` file for our server application. Inside the `server/src` directory, create the `index.js` file first. To create an `ApolloServer`, we only need to pass in our `typeDefs` and `resolvers` and call its `listen` method to start it up (under the hood, Apollo Server runs on `Express` by default). We can do that in about a dozen lines of code in the new `index.js` file:

`server/src/index.js`

```
import { ApolloServer } from "apollo-server";

import resolvers from "./graphql/resolvers.js";
import typeDefs from "./graphql/typeDefs.js";

const server = new ApolloServer({
    typeDefs,
    resolvers
});

server.listen().then(({ url }) => {
    console.log(`Server ready at ${url}`);
});
```

Just one last step! We'll need a `server:graphql` script to start up the GraphQL API. In this script, we'll set the `--ignore` option for Nodemon because we don't need to restart the server every time a record is added, updated, or removed from `db.json`. For convenience, we'll use Concurrently to create a top-level `server` script to start the REST API and GraphQL API together:

server/package.json

```
{  
  // ...  
  "scripts": {  
    "server": "concurrently -k npm:server:*",  
    "server:rest": "json-server -w db.json -p 5000 -r routes.json -q",  
    "server:graphql": "nodemon --ignore db.json ./src/index.js"  
  },  
  // ...  
}
```

If we try running `npm run server` in the terminal now, then we should see both APIs start up. By default, the GraphQL API will be available at <http://localhost:4000/> and we will access it at the `/graphql` path. We can take the GraphQL API for a quick test spin by opening another terminal tab or window and running this cURL command:

```
curl 'http://localhost:4000/graphql' -H 'Content-Type: application/json'  
-H 'Accept: application/json' --data-binary '{"query": "query { books {  
  title id } }"}' --compressed
```

The following response output should appear in the terminal:

```
{"data": {"books": [{"title": "The Hitchhiker's Guide to the  
Galaxy", "id": "1"}]}}
```

Congratulations! You are now the proud owner of a shiny, new GraphQL API.

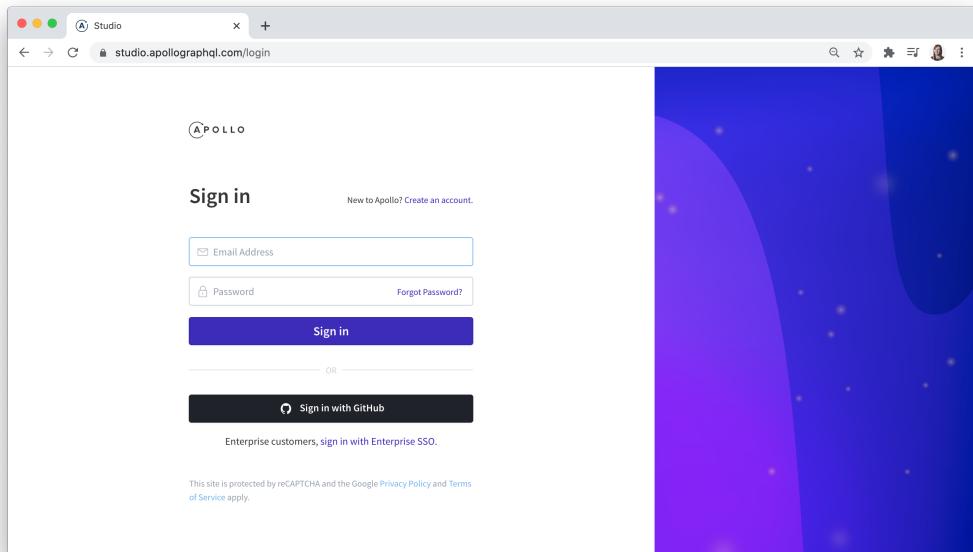
## Exploring the GraphQL API

Now that our API is up and running, it would be easier if we had a more visual way to experiment with it than cURL commands. There are many browser-based GraphQL IDEs to choose from today—we've already seen one such example when we used the embedded GraphiQL tool to try out the GitHub API. [GraphQL Playground](#) is another option, and it even comes bundled with Apollo Server. If you navigate to <http://localhost:4000/graphql> directly in your browser, you can explore your GraphQL API and run queries against it using GraphQL Playground right now.

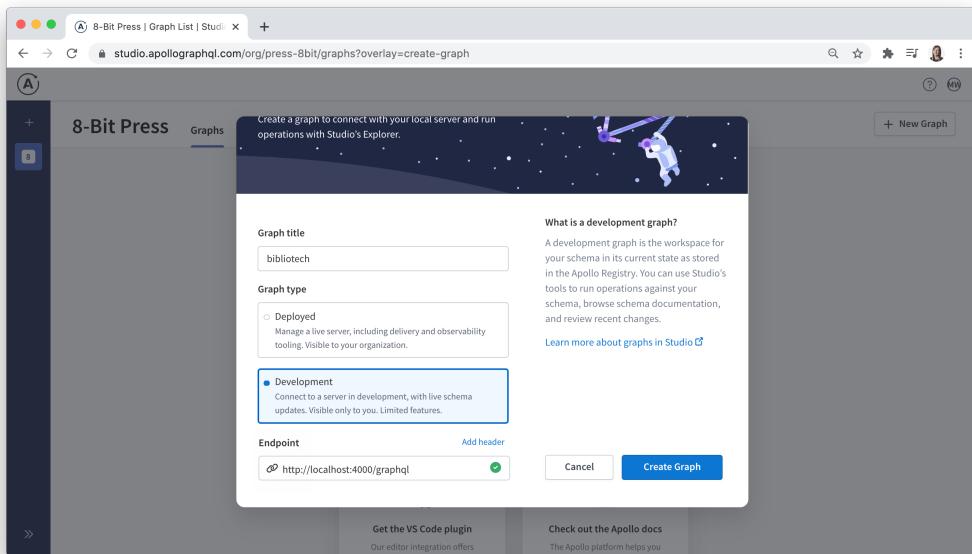
Currently, one of the most feature-rich, browser-based IDEs designed for testing GraphQL operations is [Apollo Studio Explorer](#). It includes a clickable query builder interface, operation history, and a spotlight-style search for browsing types and fields. Explorer is offered as a free feature of the [Apollo Studio](#) platform and examples throughout this book will demonstrate how to use it as a GraphQL IDE.

There are two different ways that you can access Explorer. If you would prefer to use it without creating an Apollo Studio account, you can navigate to <https://sandbox.apollo.dev> and update the sandbox URL to <http://localhost:4000/graphql> and you will be able to explore the Bibliotech GraphQL API immediately (skip to Page 21 for further directions on how to do so).

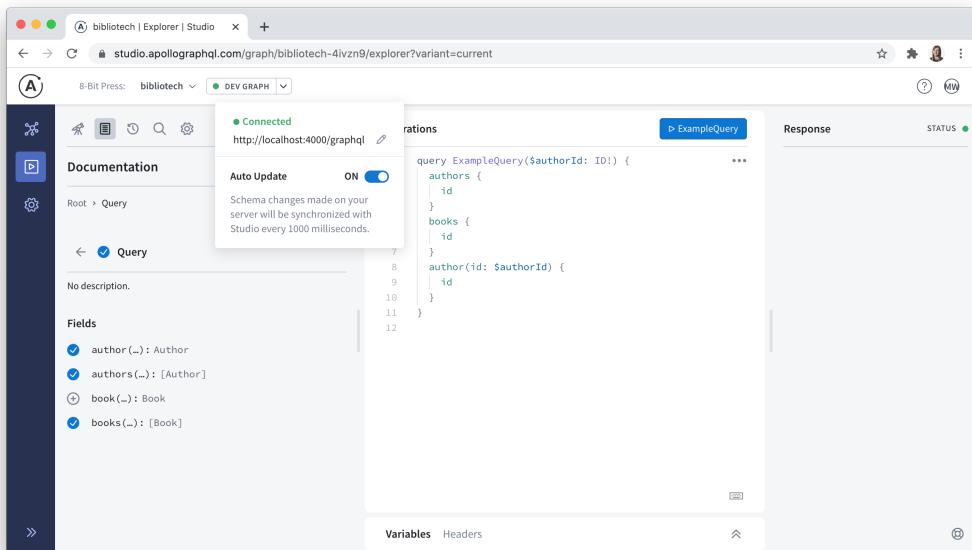
Alternatively, for a more feature-rich experience using Explorer (including a query history and additional customization settings), you can sign up for an Apollo Studio account by following these quick steps. First, navigate to <https://studio.apollographql.com/> to register for a new account:



Click the “Create an account” link and then create a new account either using your email or GitHub account. Be sure to pick the “Local development” plan. Once your account has been created, you’ll have the option to choose a “Deployed” or “Development” graph. A deployed graph can take advantage of Apollo Studio features such as its schema registry, metrics reporting, and is typically meant for team collaboration. A development graph will poll your development server for schema updates as you work. For our purposes, we will choose a development graph. Be sure to give it a unique name and set the correct endpoint:



After creating your graph, Explorer will load with a demo query operation in the editor:



The auto update option will keep Explorer in sync with the schema in your local development environment.

Next, replace the demo query in the editor with the following operation document:

```
query {
  authors {
    id
  }
}
```

Then click on **authors: [Author]** under the “Fields” subheading in the lefthand panel. Click the plus sign next to the name and books fields to automatically add them to the query (you can also type the field names into the editor if you prefer). Click and add more fields under the books field as well, and then press the “Run” button above the editor to execute the query:

The screenshot shows the Apollo Studio interface with the following details:

- Operations:** The query editor contains the following GraphQL code:
 

```
query {
  authors {
    id
    name
    books {
      id
      title
    }
  }
}
```
- Response:** The results are displayed in a table format. The table has two columns: "id" and "name" under the "authors" row, and "id" and "title" under the "books" row. One row of data is shown:
 

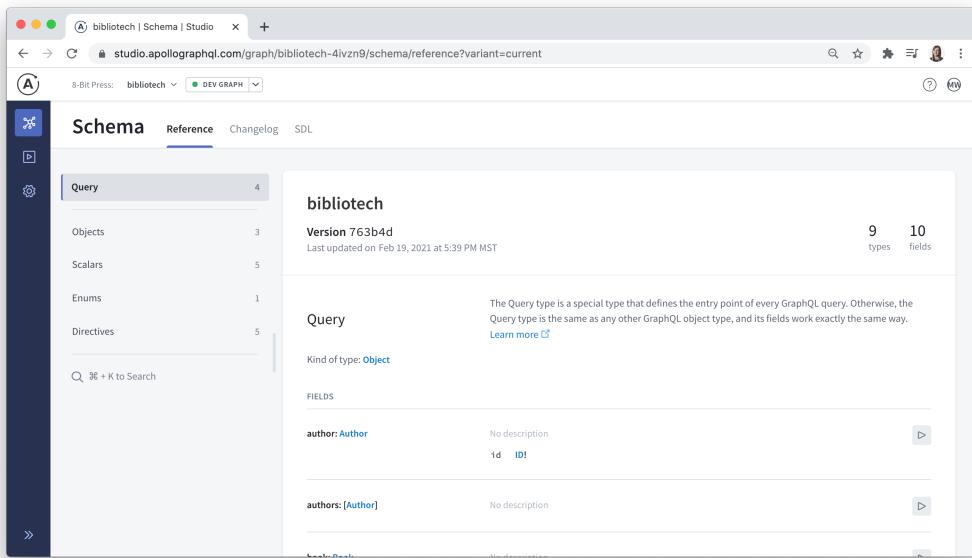
	id	name	books						
0	1	Douglas Adams	<table border="1"> <thead> <tr> <th></th> <th>id</th> <th>title</th> </tr> </thead> <tbody> <tr> <td>0</td> <td>1</td> <td>The Hitchhiker's Guide to the Galaxy</td> </tr> </tbody> </table>		id	title	0	1	The Hitchhiker's Guide to the Galaxy
	id	title							
0	1	The Hitchhiker's Guide to the Galaxy							
- Variables:** A dropdown menu is open, showing "Variables" and "Headers".
- Headers:** A dropdown menu is open, showing "Variables" and "Headers".
- Status:** The status bar at the bottom right indicates "STATUS ● 70.2ms | 125B".

In the image above, the table view has been toggled to display the results in a table format instead of the default JSON format. Keep this feature in mind when viewing complex lists of results.

Before you move on to the next chapter, be sure to try out some of Explorer’s other features, such as the spotlight-style search (accessible from the magnifying glass icon at the top of the lefthand panel). And if you signed up for an Apollo Studio account, try out some of the various settings you can configure (from the gear icon) such as dark mode and editor hints.

As a final feature to highlight, Apollo Studio also provides a convenient interface for exploring the different types in your schema. As schemas grow in size, it can become increasingly difficult to have a bird’s eye view of all of the types in it, especially if all of those type definitions are

contained within a single file. You can access an overview of all of your type definitions organized by type by navigating to the “Schema” page (it’s the first item in the lefthand navigation menu):



### Introspection: The GraphQL Magic that Makes Tools Like Explorer Possible

You may wonder how it's possible for tools such as Explorer to know so much about your GraphQL schema simply by sharing your API's endpoint. A GraphQL schema can share information about itself via *introspection*. In practice, that means that a GraphQL API exposes a special `__schema` field on the query root operation type that allows us to request information about the API itself, such as the names of its various types and fields.

You can see introspection in action by running the following query:

```
{  
  __schema {  
    types {  
      name  
      fields {  
        name  
      }  
    }  
  }  
}
```

Note that it's considered a best practice to turn off introspection in production for non-public GraphQL APIs to prevent potential bad actors from learning about the detailed inner workings of the API by sending an introspection query to it.

## Summary

We covered a lot of ground in this chapter, from learning how to write basic queries against a third-party GraphQL API, to creating type definitions for a schema of our own, and writing resolver functions to provide data for all of the fields in our API. We also learned about two named types in the GraphQL type system—Object and Scalar types—and learned how to add arguments to fields. Lastly, we saw how a visual tool like Apollo Studio Explorer can make it easy to experiment with a GraphQL API as we develop it.

In the next chapter, we'll continue to add new Object types and additional query fields to the Bibliotech API while also adopting some best practices related to GraphQL API design and development.

## Chapter 2

# More on Queries

In this chapter, we will:

- Understand how to design a GraphQL schema in a client-focused way
- Use an Apollo RESTDataSource to encapsulate data fetching from the REST API
- Add additional types and fields to the API for reviews and users
- Use operation names and variables
- Use fragments to reuse field selections across operations

## Best Practice: Design the Schema with Clients in Mind

So far we've added some basic Author and Book type definitions to a schema, along with queries to fetch authors and books individually by IDs or as a complete list. And it would be fair to say that these queries seem a bit REST-like given that they mirror what could be fetched by sending GET requests to the /authors or /books endpoints of the old Bibliotech API.

However, one of the superpowers of GraphQL is that it allows you to design your API with client use cases in mind. We typically want to prevent the types and fields in our GraphQL API from being unduly influenced by the implementation details of their backing data sources. Bibliotech will only have a single React application, so as we move forward and add new types and fields to the API we will do so with an eye for the product experiences we are trying to drive via that client.

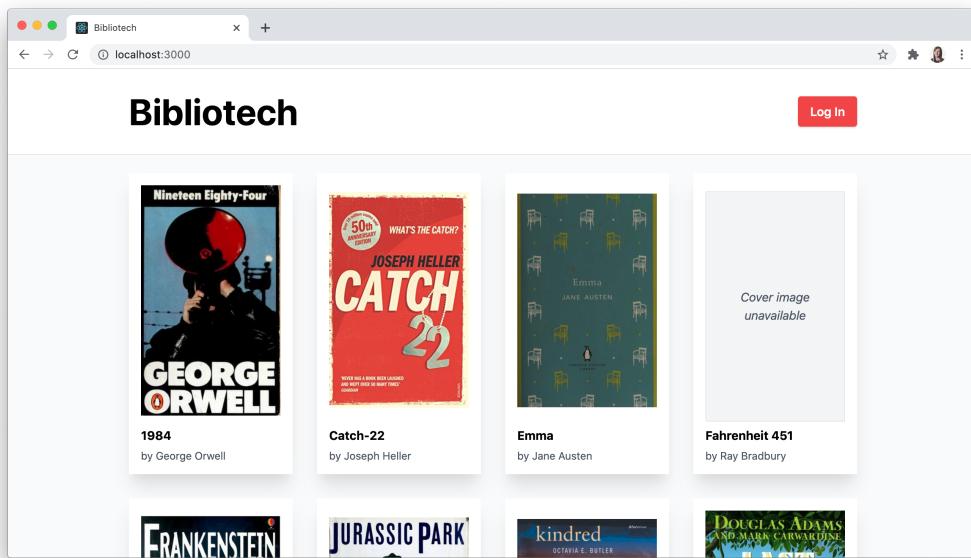
The best way to find out how to design a GraphQL API in a client-focused manner is to talk to the actual client developers who have built (or will build) the application that consumes the GraphQL API. Given that we will also be the client developers in this case, let's temporarily put on that hat and determine what the ideal shape of an API response would be when populating various views in the app with the necessary data.

To begin, let's consider some of the key features of the Bibliotech client application:

- Display a paginated list of books with their titles, authors, and cover images (if available)
- Allow users to search for books by title or author name

- Display a single book with its title, author, summary, and user ratings and reviews
- Allow users to log into the application
- Allow authenticated users to add new books to the Bibliotech catalog, save their favorite books to a personal library, and leave reviews on books
- Add new reviews and ratings in real-time to a book's review list
- Allow authenticated users to update or delete their existing review for a book

Let's take a look at the index page of the React application again:



It seems as though our existing books query will work well for this page, but we will need to add fields for the cover image and summary. Sometimes a book's cover image or summary may not be available, so these fields should be nullable. We'll also need some way to allow authenticated users to add books to Bibliotech, but we'll cover that topic in Chapter 3 when we learn about mutations.

Note that the user library page and the book search results page will have a similar grid-based book layout as the index page. Conceivably, we could reuse the books query to populate the data for the search results page by adding an argument to it for a search string, but with GraphQL it's typically considered a best practice to add finer-grained, purpose-built queries to the API. For that reason, we will end up adding separate query fields for search later on.

Next, let's examine what a single book page looks like for authenticated users:

The screenshot shows a web browser window for the BiblioTech application at localhost:3000/book/2. The page title is "Bibliotech". The main content area displays the book "The Hitchhiker's Guide to the Galaxy" by Douglas Adams. A summary text states: "After a Vogon constructor fleet destroys Earth to make way for a hyperspace bypass, the last surviving man, Arthur Dent, embarks on an interstellar adventure with his friend Ford Prefect (who, apparently, was an alien all along.)". Below the summary is a red "Remove from Library" button. To the right of the summary is the book cover art, which features a hand pointing upwards against a starry background with various celestial bodies and a planet. At the top right of the page are links for "My Library", "Add Book", and "Log out".

This screenshot shows the same BiblioTech application interface as the previous one, but it includes a "What Readers Say" section. It lists two reviews:

- Jo March** rated this book 4/5, reviewed on March 14, 2021. The review text says, "I thought it was pretty good."
- Alice Liddell** rated this book 5/5, reviewed on March 12, 2021. The review text says, "This book is out of this world."

At the bottom of the "What Readers Say" section are "Update" and "Delete" buttons. A footer quote at the bottom of the page reads: "A word after a word after a word is power." – Margaret Atwood.

The existing book query will likely be a good starting point to populate the required data on this page, though we will need to augment it with a field for the book's reviews so all of the data may be fetched with a single query. Each review will contain a creation date, rating, and review for

the book. A review must also be connected to the user who left it so their name can be displayed with it. As with books, long lists of reviews will require pagination. Lastly, we'll need some way to indicate if a currently authenticated user has already added a particular book to the library and if they have already reviewed the book.

With these observations in mind, we can now move forward and augment our GraphQL API with additional types and fields that will help support rapid development of our React application in later chapters.

## Organize Code with an Apollo Data Source

But first, it would be a good idea to pause here and wrangle our resolver code. While there are only a few dozen lines of code in the `resolvers.js` file right now, it won't take long for this file to balloon in size as we add data-fetching logic to the body of each resolver function. Further, when we handle this logic on a resolver-by-resolver basis, we miss out on the opportunity to reuse code across resolvers that fetch similar data.

Before things get out of hand, we can set up an Apollo `RESTDataSource` with methods that will encapsulate data fetching from the Bibliotech REST API. As a bonus, the `RESTDataSource` class contains several convenience methods that are designed to facilitate HTTP requests to a REST API. To use this data source, we'll need to install another package in the `server` directory:

```
npm install apollo-datasource-rest@0.11.0
```

In addition to the ready-made `RESTDataSource`, there are many community data sources available. You can even create custom data sources too. See detailed documentation about Apollo data sources [here](#):

<https://www.apollographql.com/docs/apollo-server/data/data-sources/>

Next, create a `dataSources` directory inside of `server/src/graphql`, create a `JsonServerApi.js` file in the new directory, and add the following code to it:

`server/src/graphql/dataSources/JsonServerApi.js`

```
import { RESTDataSource } from "apollo-datasource-rest";

class JsonServerApi extends RESTDataSource {
  // Data source code will go here...
}
```

```
export default JsonServerApi;
```

Above, we use the `RESTDataSource` by creating a subclass from it called `JsonServerApi`. The `RESTDataSource` expects us to define a `baseURL` property. We could add it as a hardcoded string here, but we may want to provide different versions of the Bibliotech REST API URL depending on what environment our code is running in, so we'll set it as an environment variable instead. While we're at it, we can explicitly set the `NODE_ENV` variable to `development`.

Create a `.env` file in the `server` directory and add these two variables to it:

```
NODE_ENV=development
REST_API_BASE_URL=http://localhost:5000
```

In Chapter 1, we learned that it's a best practice to turn off introspection for a non-public GraphQL API in production. Apollo Server will automatically disable introspection and GraphQL Playground when the `NODE_ENV` is set to `production`. For all other environments, these features will be available by default, but they can be manually configured in the options passed to the `ApolloServer` constructor.

We already installed the `dotenv` package in the previous chapter, so to load our environment variables we only need to update the `server:graphql` script in `package.json` to preload this package using the `-r` option:

`server/package.json`

```
{
  // ...
  "scripts": {
    "server": "concurrently -k npm:server:*",
    "server:rest": "json-server -w db.json -p 5000 -r routes.json -q",
    "server:graphql": "nodemon --ignore db.json -r dotenv/config
      ./src/index.js"
  },
  // ...
}
```

Before moving on, restart the server with `npm run server`. Next, let's set the `baseURL` property in `JsonServerApi` and make use of the `get` method from the parent `RESTDataSource` to fetch author-related data from the REST API:

server/src/graphql/dataSources/JsonServerApi.js

```
import { RESTDataSource } from "apollo-datasource-rest";

class JsonServerApi extends RESTDataSource {
  baseURL = process.env.REST_API_BASE_URL;

  getAuthorById(id) {
    return this.get(`/authors/${id}`).catch(
      err => err.message === "404: Not Found" && null
    );
  }

  async getAuthorBooks(authorId) {
    const items = await this.get(`/authors/${authorId}/books`);
    return items.map(item => item.book);
  }

  getAuthors() {
    return this.get(`/authors`);
  }
}

export default JsonServerApi;
```

The code inside these methods looks a lot like what we wrote in our resolver functions, but the `get` method handles resolving the JSON response to a JavaScript object, so we can skip calling the `json` method after fetching the data. Let's add similar methods to handle book fetching as well:

server/src/graphql/dataSources/JsonServerApi.js

```
import { RESTDataSource } from "apollo-datasource-rest";

class JsonServerApi extends RESTDataSource {
  // ...

  getBookById(id) {
    return this.get(`/books/${id}`).catch(
      err => err.message === "404: Not Found" && null
    );
  }

  async getBookAuthors(bookId) {
```

```
    const items = await this.get(`/books/${bookId}/authors`);
    return items.map(item => item.author);
}

getBooks() {
  return this.get(`/books`);
}

export default JsonServerApi;
```

Now that we have all of the data source methods we need for our existing fields, we need some way to call them from within the applicable resolver functions. To make that happen, we'll use the `dataSources` option in the `ApolloServer` constructor. The `dataSources` option is a function that returns an object containing the instantiated data source object. We can add as many data sources to this object as we want if there are additional services from which we need to fetch data for our resolvers.

Update the code in the main `index.js` file as follows:

`server/src/index.js`

```
import { ApolloServer } from "apollo-server";

import JsonServerApi from "./graphql/dataSources/JsonServerApi.js";
// ...

const server = new ApolloServer({
  typeDefs,
  resolvers,
  dataSources: () => {
    return {
      jsonServerApi: new JsonServerApi()
    };
  }
);
// ...
```

By setting the `dataSources` option, Apollo Server will add our new data source to the context for every request. In other words, the methods from the instantiated `JsonServerApi` object will be accessible from the `context` parameter in every resolver function now. Let's update our resolvers to use those methods:

server/src/graphql/resolvers.js

```
const resolvers = {
  Author: {
    books(author, args, { dataSources }, info) {
      return dataSources.jsonServerApi.getAuthorBooks(author.id);
    }
  },
  Book: {
    authors(book, args, { dataSources }, info) {
      return dataSources.jsonServerApi.getBookAuthors(book.id);
    }
  },
  Query: {
    author(root, { id }, { dataSources }, info) {
      return dataSources.jsonServerApi.getAuthorById(id);
    },
    authors(root, args, { dataSources }, info) {
      return dataSources.jsonServerApi.getAuthors();
    },
    book(root, { id }, { dataSources }, info) {
      return dataSources.jsonServerApi.getBookById(id);
    },
    books(root, args, { dataSources }, info) {
      return dataSources.jsonServerApi.getBooks();
    }
  }
};

export default resolvers;
```

Much tidier! We are even able to remove the `node-fetch` import and the `baseURL` constant from the top of `resolvers.js`. In the code above, note that we now destructure `dataSources` from the `context` parameter for each resolver and we have removed the `async` keyword from all of the existing resolvers because they no longer need to use `await` anymore. We will make further use of the `context` parameter in Chapter 6 when we add authentication and authorization to the API.

Before moving on, try running all of the author and book queries to make sure they still return the same results as before.

## Add a Review Type Definition

It's finally time to add some new types to the schema. We need to be able to query a list of related reviews with each book, so we'll create a new Object type called `Review` and then add a corresponding `reviews` field to the `Book` type. But first, as we noted earlier in the chapter, we still need to add nullable `cover` and `summary` fields to the `Book` type, so let's do that now:

*server/src/graphql/typeDefs.js*

```
import { gql } from "apollo-server";

const typeDefs = gql`  
# ...  
  
type Book {  
    id: ID!  
    authors: [Author]  
    cover: String  
    summary: String  
    title: String!  
}  
  
# ...  
`;  
  
export default typeDefs;
```

We'll also need some sample review data to query from the Bibliotech REST API, so we'll update db.json to serve that data from a new /reviews endpoint:

server/db.json

```
{
    // ...
    "bookAuthors": [
        // ...
    ],
    "reviews": [
        {
            "id": 1,
            "bookId": 1,
            "createdAt": "2021-02-20T15:02:28.308Z",
            "rating": 5,
```

```
        "userId": 1,
        "text": "This book is out of this world!"
    }
]
}
```

We don't have a user with an ID of 1 yet, but we'll add that in the next section. Let's add the `Review` type next. A rating out of five stars and a date the review was created will be guaranteed for every book, but the written comment will be optional. We will also leave the `book` field as nullable so that if a book is removed from the Bibliotech database, then a user's orphaned review for it may still be queried:

*server/src/graphql/typeDefs.js*

```
import { gql } from "apollo-server";

const typeDefs = gql`  
# ...

type Review {
  id: ID!
  book: Book
  rating: Int!
  reviewedOn: String!
  text: String
}

# ...
`;

export default typeDefs;
```

Next, we'll add a new `reviews` field to `Book` to connect books to reviews in the opposite direction:

*server/src/graphql/typeDefs.js*

```
import { gql } from "apollo-server";

const typeDefs = gql`  
# ...

type Book {
```

```
    id: ID!
    authors: [Author]
    cover: String
    reviews: [Review]
    summary: String
    title: String!
}

# ...
`;

export default typeDefs;
```

We'll also need to add a field to the `Query` type to allow single reviews to be queried by ID. This field will be helpful when we add a form in the React application that allows users to update a review (because the existing review data will need to be queried and pre-populated in the form's fields):

`server/src/graphql/typeDefs.js`

```
import { gql } from "apollo-server";

const typeDefs = gql` 
# ...

type Query {
  author(id: ID!): Author
  authors: [Author]
  book(id: ID!): Book
  books: [Book]
  review(id: ID!): Review
}
`;

export default typeDefs;
```

Now we'll focus on adding the resolvers that query a single review and that connect a book to its reviews, and vice versa. To do that, we'll add two new methods to the `JsonServerApi` data source—one that fetches a review by its ID and one that fetches all reviews for a given book ID:

server/src/graphql/dataSources/JsonServerApi.js

```
import { RESTDataSource } from "apollo-datasource-rest";

class JsonServerApi extends RESTDataSource {
  // ...

  getBookReviews(bookId) {
    return this.get(`reviews?bookId=${bookId}`);
  }

  // ...

  getReviewById(id) {
    return this.get(`reviews/${id}`).catch(
      err => err.message === "404: Not Found" && null
    );
  }
}

export default JsonServerApi;
```

Now we can update our Query resolvers to fetch a single review by ID. We'll also update the Book resolvers to use the new getBookReviews method for the reviews field and then add two resolvers for the Review type. One of those resolvers will map the createdAt time in db.json to the reviewedOn field and the other will use the getBook method to fetch the related book:

server/src/graphql/resolvers.js

```
const resolvers = {
  // ...
  Book: {
    authors(book, args, { dataSources }, info) {
      return dataSources.jsonServerApi.getBookAuthors(book.id);
    },
    reviews(book, args, { dataSources }, info) {
      return dataSources.jsonServerApi.getBookReviews(book.id);
    }
  },
  Review: {
    book(review, args, { dataSources }, info) {
      return dataSources.jsonServerApi.getBookById(review.bookId);
    },
  }
};
```

```
    reviewedOn(review, args, { dataSources }, info) {
      return review.createdAt;
    }
  },
Query: {
  // ...
  books(root, args, { dataSources }, info) {
    return dataSources.jsonServerApi.getBooks();
  },
  review(root, { id }, { dataSources }, info) {
    return dataSources.jsonServerApi.getReviewById(id);
  }
}
};

export default resolvers;
```

All other properties in a fetched review will map directly to the Review field names, so they will not require additional resolver functions here. Open your GraphQL IDE and try running the following query to confirm that a review list correctly appears under the book:

*GraphQL Query*

```
query {
  book(id: "1") {
    id
    title
    reviews {
      id
      rating
      reviewedOn
      text
    }
  }
}
```

*API Response*

```
{
  "data": {
    "book": {
      "id": "1",
```

```
"title": "The Hitchhiker's Guide to the Galaxy",
"reviews": [
  {
    "id": "1",
    "rating": 5,
    "reviewedOn": "2021-02-20T15:02:28.308Z",
    "text": "This book is out of this world!"
  }
]
```

Also, confirm that a single review is queryable by its ID:

*GraphQL Query*

```
query {
  review(id: "1") {
    id
    rating
    reviewedOn
    text
    book {
      title
    }
  }
}
```

*API Response*

```
{
  "data": {
    "review": {
      "id": "1",
      "rating": 5,
      "reviewedOn": "2021-02-20T15:02:28.308Z",
      "text": "This book is out of this world!",
      "book": {
        "title": "The Hitchhiker's Guide to the Galaxy"
      }
    }
  }
}
```

```
    }  
}
```

## Add a User Type Definition

Now we'll work through similar steps to add a `User` type to the schema. Users will be connected to the reviews they write and to the books they add to their library as favorites. We'll begin by adding sample user data to the REST API at a basic `/users` endpoint and a `/userBooks` endpoint that will handle the many-to-many relationship between users and books:

`server/db.json`

```
{  
  // ...  
  "reviews": [  
    // ...  
  ],  
  "users": [  
    {  
      "id": 1,  
      "email": "alice@email.com",  
      "name": "Alice Liddell",  
      "username": "rabbithole84"  
    }  
  ],  
  "userBooks": [  
    {  
      "id": 1,  
      "bookId": 1,  
      "createdAt": "2021-02-20T16:01:17.511Z",  
      "userId": 1  
    }  
  ]  
}
```

Note that the `createdAt` field in a `userBooks` object will help facilitate sorting these records later on. To make it easier to query the books in a user's library, we'll also add a custom route to the REST API in the `routes.json` file:

server/routes.json

```
{  
  "/authors/:authorId/books":  
    "/bookAuthors?authorId=:authorId&_expand=book",  
  "/books/:bookId/authors": "/bookAuthors?bookId=:bookId&_expand=author",  
  "/users/:userId/books": "/userBooks?userId=:userId&_expand=book"  
}
```

Next, we'll update the schema by adding a `User` type, a `reviewer` field to the `Review` type that outputs a `User`, and a field on `Query` to fetch a single user by username:

server/src/graphql/typeDefs.js

```
import { gql } from "apollo-server";  
  
const typeDefs = gql`  
  # ...  
  
  type Review {  
    id: ID!  
    book: Book  
    rating: Int!  
    reviewedOn: String!  
    reviewer: User!  
    text: String  
  }  
  
  type User {  
    id: ID!  
    email: String!  
    library: [Book]  
    name: String!  
    reviews: [Review]  
    username: String!  
  }  
  
  type Query {  
    # ...  
    user(username: String!): User  
  }  
`;
```

```
export default typeDefs;
```

It's a functional requirement for Bibliotech to delete all of a user's data (including their reviews) if their account is removed, so we make the `reviewer` field on the `Review` type non-null given that we can always expect it to be connected to some existing user.

Now we need to determine how to resolve values for all of the user-related fields we just added to the schema. We'll start by resolving the appropriate user for the `reviewer` field on the `Review` type. We need a way to fetch a user by their unique ID to do this, so we'll create a new method in our data source first:

`server/src/graphql/dataSources/JsonServerApi.js`

```
import { RESTDataSource } from "apollo-datasource-rest";

class JsonServerApi extends RESTDataSource {
  // ...

  getUserById(id) {
    return this.get(`users/${id}`).catch(
      err => err.message === "404: Not Found" && null
    );
  }

  export default JsonServerApi;
```

Next, we'll add a resolver for the `reviewer` field in `resolvers.js`:

`server/src/graphql/resolvers.js`

```
const resolvers = {
  // ...
  Review: {
    book(review, args, { dataSources }, info) {
      return dataSources.jsonServerApi.getBookById(review.bookId);
    },
    reviewedOn(review, args, { dataSources }, info) {
      return review.createdAt;
    },
    reviewer(review, args, { dataSources }, info) {
      return dataSources.jsonServerApi.getUserById(review.userId);
    }
  }
}

export default resolvers;
```

```
        },
    },
    // ...
};

export default resolvers;
```

Apollo Server will use the default resolver to map the values for the `id`, `email`, `name`, and `username` properties in a user record in `db.json` to the `User` type fields, so we only need to provide resolver functions for the `reviews` and `library` fields. To do that, we'll add two more methods to the `JsonServerApi` data source:

`server/src/graphql/dataSources/JsonServerApi.js`

```
import { RESTDataSource } from "apollo-datasource-rest";

class JsonServerApi extends RESTDataSource {
    // ...

    async getUserLibrary(userId) {
        const items = await this.get(`/users/${userId}/books`);
        return items.map(item => item.book);
    }

    getUserReviews(userId) {
        return this.get(`/reviews?userId=${userId}`);
    }
}

export default JsonServerApi;
```

The `getUserLibrary` method is implemented much like the `getAuthorBooks` method by fetching all of the user's books from the custom route we previously defined. Next, we'll add the resolvers for the `User` type:

`server/src/graphql/resolvers.js`

```
const resolvers = {
    // ...
    User: {
        library(user, args, { dataSources }, info) {
            return dataSources.jsonServerApi.getUserLibrary(user.id);
        }
    }
}
```

```
    },
    reviews(user, args, { dataSources }, info) {
      return dataSources.jsonServerApi.getUserReviews(user.id);
    }
  },
  // ...
};

export default resolvers;
```

Lastly, we need to add the resolver for the new user query. To fetch a user by their username from the REST API, we must append a query parameter to the /users endpoint. Let's add a new getUser method to the JsonServerApi data source to handle this request:

*server/src/graphql/dataSources/JsonServerApi.js*

```
import { RESTDataSource } from "apollo-datasource-rest";

class JsonServerApi extends RESTDataSource {
  // ...

  async getUser(username) {
    const [user] = await this.get(`/users?username=${username}`);
    return user;
  }

  // ...
}

export default JsonServerApi;
```

When we apply a filter to the /users endpoint using query parameters, it will return an array of users rather than a single user object, so we must destructure the first item out of the returned array and return that object from the method. Because usernames must be unique, we can be confident that there will only ever be at most one item in this array. As a final step, we can call the getUser method from the new user resolver under the Query type:

*server/src/graphql/resolvers.js*

```
const resolvers = {
  // ...
  Query: {
```

```
// ...
review(root, { id }, { dataSources }, info) {
  return dataSources.jsonServerApi.getReviewById(id);
},
user(root, { username }, { dataSources }, info) {
  return dataSources.jsonServerApi.getUser(username);
}
};

export default resolvers;
```

We're ready to run some queries that include user data now. Let's first run the previous query for a single book with reviews but also include a `reviewer` field this time:

#### *GraphQL Query*

```
query {
  book(id: "1") {
    id
    title
    reviews {
      rating
      text
      reviewer {
        name
      }
    }
  }
}
```

#### *API Response*

```
{
  "data": {
    "book": {
      "id": "1",
      "title": "The Hitchhiker's Guide to the Galaxy",
      "reviews": [
        {
          "rating": 5,
          "text": "This book is out of this world!",
        }
      ]
    }
  }
}
```

```
        "reviewer": {
          "name": "Alice Liddell"
        }
      }
    ]
  }
}
```

Now let's try a larger query that fetches fields for every Object type we've defined in our schema so far. We'll start by querying a single user by their username and we'll include all of the books they've added to their library with author details, as well as all of the reviews that they have written including information about the book that was reviewed:

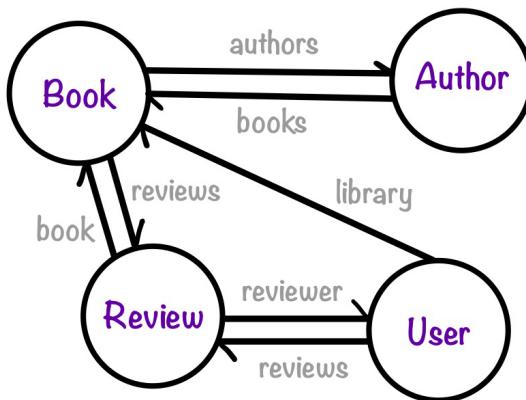
#### *GraphQL Query*

```
query {
  user(username: "rabbithole84") {
    id
    email
    library {
      id
      title
      authors {
        name
      }
    }
    name
    reviews {
      id
      book {
        id
        title
      }
      rating
      text
    }
    username
  }
}
```

*API Response*

```
{  
  "data": {  
    "user": {  
      "id": "1",  
      "email": "alice@email.com",  
      "library": [  
        {  
          "id": "1",  
          "title": "The Hitchhiker's Guide to the Galaxy",  
          "authors": [  
            {  
              "name": "Douglas Adams"  
            }  
          ]  
        }  
      ],  
      "name": "Alice Liddell",  
      "reviews": [  
        {  
          "id": "1",  
          "book": {  
            "id": "1",  
            "title": "The Hitchhiker's Guide to the Galaxy"  
          },  
          "rating": 5,  
          "text": "This book is out of this world!"  
        }  
      ],  
      "username": "rabbithole84"  
    }  
  }  
}
```

With just a few Object types defined in our schema, we can already query data from our API in such a way that expresses the complex relationships between the different nodes in our data graph along with the edges that connect them. The following diagram helps visualize those relationships in a graph format:



## Best Practice: Use Operation Names and Variables

Before wrapping up this chapter and moving on to mutations, there are some important features of GraphQL operation document syntax we should cover. The first is that we're not limited to running anonymous operations using the `query` keyword alone. It's usually considered a best practice to give your operations a distinct and descriptive name too<sup>1</sup>. For example:

*GraphQL Query*

```

query GetBook {
  book(id: "1") {
    id
    title
  }
}
  
```

Uniquely identifying the operations performed by clients is useful because it helps with tracing and debugging GraphQL API requests in your observability tooling. Named operations are also useful inside of the non-sandbox version of Explorer because you will have a better idea of which operations you have run recently by looking at the “Run history” list in the lefthand panel (without operation names, all operations will be listed as `(unnamed operation)`).

The second important syntactical feature is query *variables*<sup>2</sup>. So far, we have hardcoded argument values when we send a query for an individual author, book, or user. However, we can parameterize a query with variables to make it easier to reuse and avoid dynamically building query strings with interpolated argument values in a real client. We could add a variable for a book ID to the previous `GetBook` query as follows:

<sup>1</sup><https://spec.graphql.org/Jun2018/#sec-Named-Operation-Definitions>

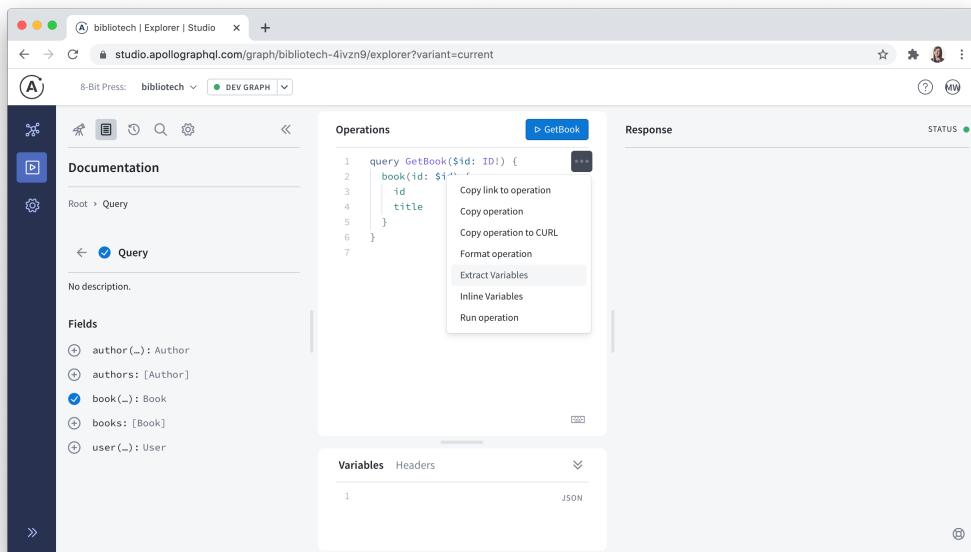
<sup>2</sup><https://spec.graphql.org/Jun2018/#sec-Language.Variables>

### GraphQL Query

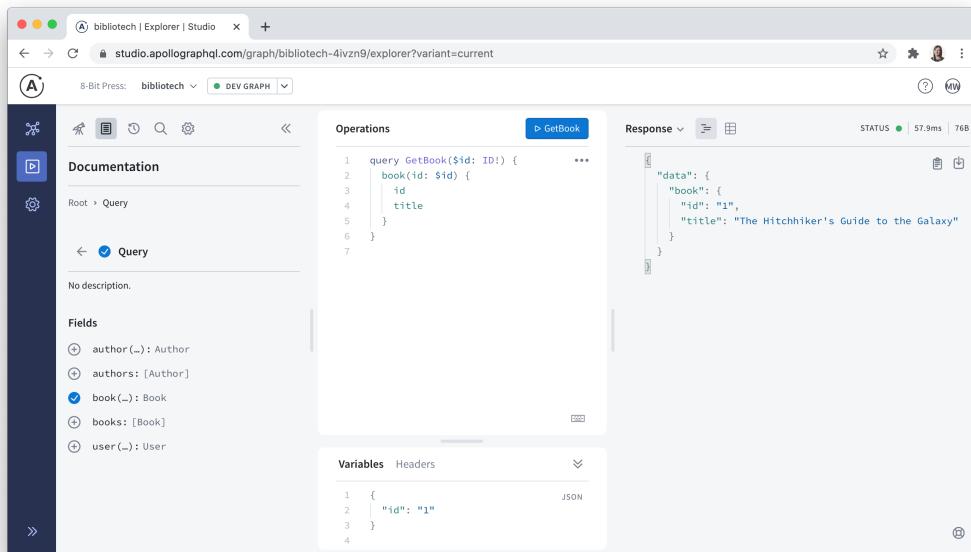
```
query GetBook($id: ID!) {
  book(id: $id) {
    id
    title
  }
}
```

When variables are added to an operation they are added in parentheses before the opening curly bracket, they are denoted with a \$ character, and they must have a type specified as well. The variable can then be substituted for argument values as needed throughout the operation document.

To run a query with variables in Explorer, we can use the “Variables” panel below the main editor and write them in JSON format. You can automatically extract all of the variables as JSON in this panel by choosing “Extract Variables” from the main editor menu:



With the variables extracted, you can fill in real values and submit your query as you did before:



## DRYer Operations with Fragments

A final advanced feature of GraphQL operation syntax that we'll explore in this chapter is called a *fragment*<sup>3</sup>. Fragments allow us to reuse common selections of fields across multiple queries. For example, in our previous complex query for a single user with their library books and their reviews with book details, we repeated both the `id` and `title` fields for each connected book. We could have instead defined a fragment above or below the query in the editor like this:

### GraphQL Fragment

```

fragment BookFields on Book {
  id
  title
}

```

And then replaced those fields under the `library` field and `book` field using the fragment name preceded by the spread operator:

<sup>3</sup><https://spec.graphql.org/June2018/#sec-Language.Fragments>

### GraphQL Query

```
query GetUser($username: String!) {
  user(username: $username) {
    id
    email
    library {
      ...BookFields
      authors {
        name
      }
    }
    name
    reviews {
      id
      book {
        ...BookFields
      }
      rating
      text
    }
    username
  }
}
```

### Query Variables

```
{
  "username": "rabbithole84"
}
```

Later in this book when we work on the React application, we'll use a fragment much like this one to create a reusable grouping of fields for books.

## Summary

In this chapter, we continued to build out the GraphQL API for Bibliotech with an eye for adding types and fields that will support client development, rather than simply mirroring back-end implementation details. We learned how Apollo data sources can help maximize code reuse in resolver functions and we also added two new Object types called `Review` and `User` to the schema. We then followed some operation-related best practices by naming our operations, using query variables, and writing fragments of field selections to be reused across queries.

In the next chapter, we build even more powerful features into our GraphQL API by adding mutation operations that will finally allow us to create, update, and delete data for authors, books, reviews, and users.

## Chapter 3

# Mutating Data

In this chapter, we will:

- Write data via a GraphQL API using mutations
- Add mutations to create authors and books
- Using an Input Object type to handle complex field arguments
- Add mutations to create reviews and handle errors when a user tries to review the same book twice
- Add mutations that allow users to update and delete their reviews
- Sign up a new user with a mutation and validate unique username and email values
- Add mutations to add and remove books from a user's library

## Mutations: How We Write Data via a GraphQL API

After some considerable practice reading data using GraphQL queries, it's time to move on to writing data. In GraphQL speak, that means that we will be creating and running mutation operations. Mutation syntax largely resembles what we have become accustomed to with queries, but we'll define mutations on the `mutation` root operation type, or in other words, as fields on the `Mutation` Object type.

As with queries, when we define mutations for a GraphQL API we want to keep client use cases in mind. That means that as we add fields to the `Mutation` Object type, we will focus on creating mutations that directly support the actions users want to take inside of the React application, rather than blindly replicating what can be done with the Bibliotech REST API endpoints (remember, we want to focus on designing around product experiences and keep those back-end implementation details out of the schema).

And as a further best practice, we will strive to create finer-grained mutations that facilitate specific user actions. And to that end, by the conclusion of this chapter we will have added individual mutations that create authors, books, reviews, and users; update or delete reviews; add or remove authors from a book, and; add or remove books from a user's library.

### Mo Mutations Mo Problems?

If the thought of adding a large number of highly specific mutations (and queries) to a GraphQL API makes you concerned that the schema may become bloated over time with unused fields as client needs evolve, then fear not! GraphQL is designed to support an API's continuous evolution (without versioning) and there's a built-in way to flag deprecate fields.

GraphQL-focused observability tools also help you understand how fields in your API are being used, what clients are using them, and how recently they have been used so that deprecated fields may be safely retired from the API.

## Create an Author

Our first mutation will allow us to create a new author. To do that we must update the type definitions to include the `Mutation` Object type and add the `createAuthor` mutation as its first field. This mutation will have a single name argument that takes a non-null `String` and the field output type will be a non-null `Author` (meaning that after an author is created, we can expect to receive data about that new author back in the response):

`server/src/graphql/typeDefs.js`

```
import { gql } from "apollo-server";

const typeDefs = gql`  
# ...  
  
type Mutation {  
  createAuthor(name: String!): Author!  
}  
;  
  
export default typeDefs;
```

Note that using non-null mutation arguments can help clients understand exactly what data is mandatory when they submit a mutation operation and also help eliminate the need to do runtime validation manually when a specific argument is required to complete a given mutation. Next, we need to add a new method to the `JsonServerApi` data source. For the first time, we will use the `post` method exposed in its parent class to create a new user:

server/src/graphql/dataSources/JsonServerApi.js

```
import { RESTDataSource } from "apollo-datasource-rest";

class JsonServerApi extends RESTDataSource {
  // ...

  createAuthor(name) {
    return this.post("/authors", { name });
  }
}

export default JsonServerApi;
```

When we send this POST request to the running JSON Server will see that a new author record has been written to the db.json file, and thus will be queryable by the GraphQL API. To finish wiring up this mutation, we'll add a `Mutation` key to the `resolvers` object and call the new method inside a `createAuthor` resolver, passing through the `name` argument's value:

server/src/graphql/resolvers.js

```
const resolvers = {
  // ...
  Query: {
    // ...
  },
  Mutation: {
    createAuthor(root, { name }, { dataSources }, info) {
      return dataSources.jsonServerApi.createAuthor(name);
    }
  }
}

export default resolvers;
```

To run a mutation we must now use the `mutation` keyword instead of the `query` keyword in the operation document, but the rest of the syntax will be the same. We'll add `id` and `name` fields as selections so that we receive those values in the response after the mutation completes. Try running the following mutation from Explorer to create a new book:

### GraphQL Mutation

```
mutation CreateAuthor($name: String!) {
  createAuthor(name: $name) {
    id
    name
  }
}
```

### Mutation Variables

```
{
  "name": "Ursula K. Le Guin"
}
```

### API Response

```
{
  "data": {
    "createAuthor": {
      "id": "2",
      "name": "Ursula K. Le Guin"
    }
  }
}
```

## Create a Book with an Input Type

Onward to books! We'll follow similar steps to add a mutation to create a new book now. However, adding a new book to Bibliotech is a bit more complex because a new book must have a title and it also may have any number of authors assigned (based on the authors' IDs) and an optional cover image URL. Adding a `createBook` mutation with those four arguments would look like this:

`server/src/graphql/typeDefs.js`

```
import { gql } from "apollo-server";

const typeDefs = gql`  
# ...  
  
type Mutation {
```

```
createAuthor(name: String!): Author!
createBook(
  authorIds: [ID]
  cover: String
  summary: String
  title: String!
): Book!
}
`;

export default typeDefs;
```

This approach will certainly work when handling multiple arguments, but there's a tidier way that we can pass a complex value into a field called an *Input Object*—the third kind of named type that we will incorporate into our schema. We create an Input Object type much as we do a regular Object type with any combination of nullable, non-null, and list fields, but we use the `input` keyword in front of it instead:

`server/src/graphql/typeDefs.js`

```
import { gql } from "apollo-server";

const typeDefs = gql` 
# ...

input CreateBookInput {
  authorIds: [ID]
  cover: String
  summary: String
  title: String!
}

# ...
`;

export default typeDefs;
```

We can then rewrite the `createBook` mutation as follows:

server/src/graphql/typeDefs.js

```
import { gql } from "apollo-server";

const typeDefs = gql`  
  # ...  
  
  type Mutation {  
    createAuthor(name: String!): Author!  
    createBook(input: CreateBookInput!): Book!  
  }  
;  
  
export default typeDefs;
```

Next, we need to add a `createBook` method to the data source for use in the corresponding resolver. This method will be more complex than the `createAuthor` one is because we need to create the book object in `db.json` and also create `bookAuthor` objects to join the book to all of its authors:

server/src/graphql/dataSources/JsonServerApi.js

```
import { RESTDataSource } from "apollo-datasource-rest";  
  
class JsonServerApi extends RESTDataSource {  
  // ...  
  
  async createBook({ authorIds, cover, summary, title }) {  
    const book = await this.post("/books", {  
      ...(cover && { cover }),  
      ...(summary && { summary }),  
      title  
    });  
  
    if (authorIds?.length) {  
      await Promise.all(  
        authorIds.map(authorId =>  
          this.post("/bookAuthors", {  
            authorId: parseInt(authorId),  
            bookId: book.id  
          })
      );
    }
  };
}
```

```
    }

    return book;
}

export default JsonServerApi;
```

Now we'll set up the `createBook` resolver, passing through the `input` argument directly into the data source method:

`server/src/graphql/resolvers.js`

```
const resolvers = {
  // ...
  Mutation: {
    createAuthor(root, { name }, { dataSources }, info) {
      return dataSources.jsonServerApi.createAuthor(name);
    },
    createBook(root, { input }, { dataSources }, info) {
      return dataSources.jsonServerApi.createBook(input);
    }
  }
}

export default resolvers;
```

Before moving on, try running the new mutation to confirm that it works as expected:

*GraphQL Mutation*

```
mutation CreateBook($input: CreateBookInput!) {
  createBook(input: $input) {
    id
    cover
    title
  }
}
```

*Mutation Variables*

```
{
  "input": {
```

```
        "authorIds": ["2"],
        "cover": "http://covers.openlibrary.org/b/isbn/038051284X-L.jpg",
        "title": "The Dispossessed"
    }
}
```

*API Response*

```
{
  "data": {
    "createBook": {
      "id": "2",
      "cover": "http://covers.openlibrary.org/b/isbn/038051284X-L.jpg",
      "title": "The Dispossessed"
    }
  }
}
```

## Create a Review with Custom Error Handling

We can create authors and books now, but we also need a way for user's to add new reviews to books. Unlike with our previous mutations, there's an additional consideration when adding new reviews to ensure that each user can only submit one review per book. And to enhance user experience in the client application, we will provide a custom message in the GraphQL response that describes the runtime error so users may better understand why their request failed.

As a first step, we'll add a `CreateReviewInput` Input Object and `createReview` field on the `Mutation` type:

`server/src/graphql/typeDefs.js`

```
import { gql } from "apollo-server";

const typeDefs = gql` 
  # ...

  input CreateReviewInput {
    bookId: ID!
    rating: Int!
    reviewerId: ID!
    text: String
  }
}
```

```
}

# ...

type Mutation {
  # ...
  createReview(input: CreateReviewInput!): Review!
}
^;

export default typeDefs;
```

In the code above, note that we don't include a `reviewedOn` field in the `CreateReviewInput` because the time will be calculated automatically on the server instead of depending on the client to send this value. To that end, we'll create a new method in the `JsonServerApi` data source to facilitate review creation. Inside of this method, we will throw an error if a user attempts to create a second review for a book. To do that, we have to check the `/reviews` endpoint to see if a review already exists with `bookId` and `userId` field values that match the review to be created. If we find an existing review, then we'll throw an error so the mutation does not complete.

Before we proceed with writing the code, we should explore the topic of error handling in GraphQL responses first. If you have previously submitted an operation with an incorrect field name in the document, then you may have already encountered GraphQL's approach to handling errors in responses. When a response contains errors, information about that error will be returned under an `errors` key in the response. For example, if you try to send an `author` query with an erroneous field added to it, then you will see this output:

```
{
  "errors": [
    {
      "message": "Cannot query field \"firstName\" on type \"Author\".",
      "locations": [
        {
          "line": 3,
          "column": 5
        }
      ],
      "extensions": {
        "code": "GRAPHQL_VALIDATION_FAILED",
        "exception": {
          "stacktrace": [
            // ...
          ]
        }
      }
    }
  ]
}
```

```
        ]
    }
}
],
  "data": null
}
```

There are a few interesting things to note in this response. First, at a minimum, a GraphQL response that contains any errors will list each instance of an error under the `errors` key and each error will have a `message` key containing some information about what went wrong. Optionally, a `locations` key may be included for each error to indicate where the error occurred in the GraphQL document.<sup>1</sup>

An `extensions` key may also be included with additional information about the error. Apollo Server will provide us with an `error code` and an `exception` with a stack trace for the error under this key. Note that for security reasons, stack traces will be unavailable in the response if Apollo Server's `debug` option is set to `false` or the `NODE_ENV` environment variable is set to `test` or `production`.

A final noteworthy point about this example is that there are both `errors` and `data` keys in the response. While the `data` value is `null` here, GraphQL has a unique feature in that it is possible to send responses that contain errors and partial data depending on what kind of error occurred.

We could throw a standard JavaScript `Error` inside of our data source method to trigger an error response, but Apollo Server also has several custom error types built in to help improve communication to clients about what kind of error occurred. These errors include an `AuthenticationError`, a `ForbiddenError`, a `UserInputError`, and a general `ApolloError`. Using these errors will dictate what value is provided for the `code` key in the error response and also allow you to add additional keys under the `exception` if needed.

For our purposes, we will import and use the `ForbiddenError` to warn clients that the request failed because the user already created a review for that book:

`server/src/graphql/dataSources/JsonServerApi.js`

```
import { ForbiddenError } from "apollo-server";
import { RESTDataSource } from "apollo-datasource-rest";

class JsonServerApi extends RESTDataSource {
  // ...

  async createReview({ bookId, rating, reviewerId, text }) {
```

---

<sup>1</sup><https://spec.graphql.org/June2018/#sec-Errors>

```
const existingReview = await this.get(`
  `/reviews?bookId=${bookId}&userId=${reviewerId}`

);

if (existingReview.length) {
  throw new ForbiddenError("Users can only submit one review per
    book");
}

return this.post("/reviews", {
  ...(text && { text }),
  bookId: parseInt(bookId),
  createdAt: new Date().toISOString(),
  rating,
  userId: parseInt(reviewerId)
});
}

export default JsonServerApi;
```

Next, we'll add the `createReview` mutation in `resolvers.js`:

`server/src/graphql/resolvers.js`

```
const resolvers = {
  // ...
  Mutation: {
    // ...
    createBook(root, { input }, { dataSources }, info) {
      return dataSources.jsonServerApi.createBook(input);
    },
    createReview(root, { input }, { dataSources }, info) {
      return dataSources.jsonServerApi.createReview(input);
    }
  }
}

export default resolvers;
```

Let's take the new `createReview` mutation for a spin. Try running the operation:

*GraphQL Mutation*

```
mutation CreateReview($input: CreateReviewInput!) {
  createReview(input: $input) {
    id
    rating
    text
    reviewedOn
    book {
      title
    }
    reviewer {
      name
    }
  }
}
```

*Mutation Variables*

```
{
  "input": {
    "bookId": "2",
    "rating": 5,
    "reviewerId": "1",
    "text": "What a page turner!"
  }
}
```

*API Response*

```
{
  "data": {
    "createReview": {
      "id": "2",
      "rating": 5,
      "text": "What a page turner!",
      "reviewedOn": "2021-02-22T11:37:48.308Z",
      "book": {
        "title": "The Dispossessed"
      },
      "reviewer": {
        "name": "Alice Liddell"
      }
    }
  }
}
```

```
        }
    }
}
```

Now try running the same mutation a second time. You will see the custom "Users can only submit one review per book" error message in the response, as well as the FORBIDDEN code. GraphQL features such as these will make it much easier to design better user experiences around error handling when we build the React application in later chapters.

## Update or Delete a Review

The new Bibliotech React application will also allow users to update and delete reviews that they previously created, so in this section, we'll add mutations for both. The `updateReview` mutation will allow users to update both the rating and the review text. Particularly when adding update-related mutations to a schema, it's important to pause and consider what the client use cases for the mutations will be. It's often wise to avoid generic catch-all mutations to facilitate any variety of updates a client may need to run on an existing resource in a database. Instead, it's better to add finer-grained mutations that are tailored around specific use cases.

For the `updateReview` mutation, it will always be the case that the user is prompted to update both the `rating` and the optional `text` field together, so it will make sense to handle both of these values in a single mutation rather than two mutations that allow clients to send updated values for each field individually. We'll need an `UpdateReviewInput` type to support this mutation with non-null `id` and `rating` fields and a nullable `text` field. We'll also add a `deleteReview` mutation that accepts a single `id` argument to identify the review to be deleted and that outputs the same ID value to indicate if the deletion operation was successful:

`server/src/graphql/typeDefs.js`

```
import { gql } from "apollo-server";

const typeDefs = gql`  
# ...  
  
input UpdateReviewInput {  
  id: ID!  
  rating: Int!  
  text: String  
}  
  
# ...
```

```
type Mutation {
  # ...
  deleteReview(id: ID!): ID!
  updateReview(input: UpdateReviewInput!): Review!
}

;

export default typeDefs;
```

Next, we need to add some new methods to the `JsonServerApi` data source that make `patch` and `delete` requests to the Bibliotech REST API:

`server/src/graphql/dataSources/JsonServerApi.js`

```
// ...

class JsonServerApi extends RESTDataSource {
  // ...

  updateReview({ id, rating, text }) {
    return this.patch(`reviews/${id}`, {
      rating,
      ...(text && { text })
    });
  }

  async deleteReview(id) {
    await this.delete(`/reviews/${id}`);
    return id;
  }
}

export default JsonServerApi;
```

Lastly, we'll update our resolvers once again:

`server/src/graphql/resolvers.js`

```
const resolvers = {
  // ...
  Mutation: {
```

```
// ...
createReview(root, { input }, { dataSources }, info) {
  return dataSources.jsonServerApi.createReview(input);
},
deleteReview(root, { id }, { dataSources }, info) {
  return dataSources.jsonServerApi.deleteReview(id);
},
updateReview(root, { input }, { dataSources }, info) {
  return dataSources.jsonServerApi.updateReview(input);
}
}

export default resolvers;
```

Let's try both new mutations out now. First, we'll update the review we created in the last section with a new rating and revised text:

#### *GraphQL Mutation*

```
mutation UpdateReview($input: UpdateReviewInput!) {
  updateReview(input: $input) {
    id
    rating
    text
  }
}
```

#### *Mutation Variables*

```
{
  "input": {
    "id": "2",
    "rating": 4,
    "text": "It was pretty good."
  }
}
```

#### *API Response*

```
{
  "data": {
```

```
  "updateReview": {
    "id": "2",
    "rating": 4,
    "text": "It was pretty good."
  }
}
```

Now let's try deleting the same review. Because we are only returning an ID Scalar type from this field, we won't provide any additional field selections for the response:

#### *GraphQL Mutation*

```
mutation DeleteReview($id: ID!) {
  deleteReview(id: $id)
}
```

#### *Mutation Variables*

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

#### *API Response*

```
{
  "data": {
    "deleteReview": "2"
  }
}
```

At this point, the thought may have crossed your mind that we currently have a gaping security hole in our API where users could potentially create, update, or delete reviews the belong to other users. In Chapter 6, we'll lock this API down and ensure that users are authenticated and authorized to handle only their book reviews before executing these (and several other) mutations.

## **Sign Up a New User**

The final kind of resource to create via the API is a new user. Users must be uniquely identified by their username and email address, so the data source method must enforce this requirement and throw an error if a user tries to select a username that's already in use or if they try to sign

up with an email already assigned to another user. As usual, we'll start by updating the type definitions:

*server/src/graphql/typeDefs.js*

```
import { gql } from "apollo-server";

const typeDefs = gql`  
# ...  
  
input SignUpInput {  
  email: String!  
  name: String!  
  username: String!  
}  
  
# ...  
  
type Mutation {  
  # ...  
  signUp(input: SignUpInput!): User!  
  # ...  
}  
`;  
  
export default typeDefs;
```

To enforce unique usernames and email addresses, we'll create a utility in the `JsonServerApi` class that queries the `/users` endpoint for users with the submitted values. If a user is found matching either value, we'll throw a `UserInputError` this time:

*server/src/graphql/dataSources/JsonServerApi.js*

```
import { ForbiddenError, UserInputError } from "apollo-server";  
import { RESTDataSource } from "apollo-datasource-rest";  
  
class JsonServerApi extends RESTDataSource {  
  baseURL = process.env.REST_API_BASE_URL;  
  
  async checkUniqueUserData(email, username) {  
    const res = await Promise.all([  
      this.get(`/users?email=${email}`),  
      this.get(`/users?username=${username}`)  
    ]);  
    // ...  
  }  
}
```

```
]);
const [existingEmail, existingUsername] = res;

if (existingEmail.length) {
  throw new UserInputError("Email is already in use");
} else if (existingUsername.length) {
  throw new UserInputError("Username already in use");
}

// ...
}

export default JsonServerApi;
```

While updating `JsonServerApi`, we'll also add a new `signUp` method that calls `checkUniqueUserData` and creates a new user if no errors are thrown:

`server/src/graphql/dataSources/JsonServerApi.js`

```
// ...

class JsonServerApi extends RESTDataSource {
// ...

async signUp({ email, name, username }) {
  await this.checkUniqueUserData(email, username);
  return this.post("/users", {
    email,
    name,
    username
  });
}

// ...
}

export default JsonServerApi;
```

Lastly, we can use the new `signUp` method in the a new `signUp` mutation resolver:

server/src/graphql/resolvers.js

```
const resolvers = {
  // ...
  Mutation: {
    // ...
    signUp(root, { input }, { dataSources }, info) {
      return dataSources.jsonServerApi.signUp(input);
    },
    // ...
  }
}

export default resolvers;
```

With this code in place, we can try signing up a new user:

*GraphQL Mutation*

```
mutation SignUp($input: SignUpInput!) {
  signUp(input: $input) {
    id
    email
    name
    username
  }
}
```

*Mutation Variables*

```
{
  "input": {
    "email": "jo@email.com",
    "name": "Jo March",
    "username": "bookworm68"
  }
}
```

*API Response*

```
{
  "data": {
```

```
    "signUp": {
      "id": "2",
      "email": "jo@email.com",
      "name": "Jo March",
      "username": "bookworm68"
    }
  }
}
```

Before moving on, try signing up a new user with an existing username or email address to confirm that an error is thrown as expected.

## Update a User's Library

The final two mutations that we add in this chapter will allow us to add and remove books to and from a user's library. If you recall the design of the single book page, there is a button visible to authenticated users that allows them to toggle whether the book is stored in their library. While we could create a single generic mutation to handle either task (and perhaps also update other user fields), we will instead opt for creating purpose-built mutations to support each specific user action in the client application.

To run either mutation, we will need to submit a list of the book IDs that should be added to or removed from a user's library, as well as the ID of the user in question so that the appropriate `userBooks` record may be created or deleted. The supporting Input Object type will be identical for both mutations, so we will create a single `UpdateLibraryBooksInput` type to use for both the `addBooksToLibrary` and `removeBooksFromLibrary` mutations:

`server/src/graphql/typeDefs.js`

```
import { gql } from "apollo-server";

const typeDefs = gql` 
# ...

input UpdateLibraryBooksInput {
  bookIds: [ID!]!
  userId: ID!
}

# ...

type Mutation {
```

```
    addBooksToLibrary(input: UpdateLibraryBooksInput!): User!
    # ...
    removeBooksFromLibrary(input: UpdateLibraryBooksInput!): User!
    # ...
}
`;

export default typeDefs;
```

When sharing Input Object types between mutations, it's important to carefully consider whether the schema may evolve in such a way that would make it awkward for the queries or mutations that rely on it to continue sharing this type. For some cases, it may ultimately make more sense to create separate Input Object types to facilitate schema evolvability.

Next, we'll add data source methods for each library-related mutation. It wouldn't make sense to add the same book to the user's library twice, so the `addBooksToLibrary` method will query the `/userBooks` endpoint of the Bibliotech REST API to see if any records currently exist for each book the user wants to add to their library, filter out any duplicates, and then add records for the new library books:

`server/src/graphql/dataSources/JsonServerApi.js`

```
// ...

class JsonServerApi extends RESTDataSource {
    // ...

    async addBooksToLibrary({ bookIds, userId }) {
        const response = await Promise.all(
            bookIds.map(bookId =>
                this.get(`/userBooks/?userId=${userId}&bookId=${bookId}`)
            )
        );
        const existingUserBooks = response.flat();
        const newBookIds = bookIds.filter(
            bookId => !existingUserBooks.find(book => book.id ===
                parseInt(bookId))
        );

        await Promise.all(
```

```
    bookIds.map(bookId =>
      this.post("/userBooks", {
        bookId: parseInt(bookId),
        createdAt: new Date().toISOString(),
        userId: parseInt(userId)
      })
    );
  );

  return this.get(`/users/${userId}`);
}

// ...
}

export default JsonServerApi;
```

The `removeBooksFromLibrary` method will also ensure that the books exist in the user's library before deleting them:

*server/src/graphql/dataSources/JsonServerApi.js*

```
// ..

class JsonServerApi extends RESTDataSource {
  // ...

  async removeBooksFromLibrary({ bookIds, userId }) {
    const response = await Promise.all(
      bookIds.map(bookId =>
        this.get(`/userBooks/?userId=${userId}&bookId=${bookId}`)
      )
    );
    const existingUserBooks = response.flat();

    await Promise.all(
      existingUserBooks.map(({ id }) => this.delete(`/userBooks/${id}`))
    );

    return this.get(`/users/${userId}`);
  }
}
```

```
// ...
}

export default JsonServerApi;
```

Lastly, we'll create the resolvers for the new mutations:

*server/src/graphql/resolvers.js*

```
const resolvers = {
  // ...
  Mutation: {
    addBooksToLibrary(root, { input }, { dataSources }, info) {
      return dataSources.jsonServerApi.addBooksToLibrary(input);
    },
    // ...
    removeBooksFromLibrary(root, { input }, { dataSources }, info) {
      return dataSources.jsonServerApi.removeBooksFromLibrary(input);
    },
    // ...
  }
}

export default resolvers;
```

Now we're set to add a new book to an existing user's library:

*GraphQL Mutation*

```
mutation AddBooksToLibrary($input: UpdateLibraryBooksInput!) {
  addBooksToLibrary(input: $input) {
    name
    library {
      title
    }
  }
}
```

*Mutation Variables*

```
{
  "input": {
```

```
        "bookIds": ["2"],
        "userId": "1"
    }
}
```

#### API Response

```
{
  "data": {
    "addBooksToLibrary": {
      "name": "Alice Liddell",
      "library": [
        {
          "title": "The Hitchhiker's Guide to the Galaxy"
        },
        {
          "title": "The Dispossessed"
        }
      ]
    }
  }
}
```

Before wrapping up, try running the `removeBooksFromLibrary` mutation to ensure you can also remove the book that was just added to the same user's library.

## Summary

At the conclusion of this chapter, our GraphQL API is now capable of both reading and writing data with the addition of several fields on the root `Mutation` type. Clients can use mutation operations to create authors, books, reviews, and users. They can also update and delete reviews, and they can add and remove books from a user's library. We used Input Object types to structure complex argument configurations for mutations, and we saw how to throw errors at runtime using the `ForbiddenError` and `UserInputError` from Apollo Server.

In the next chapter, we will cover the remaining three named types available in the GraphQL type system, and while doing so, will add pagination to our existing queries and also add search functionality to the Bibliotech API.

## Chapter 4

# Pagination and Search Queries

In this chapter, we will:

- Use an `Enum` type to categorize books by genre
- Explore different pagination options for GraphQL APIs
- Add pagination to several existing fields in the schema
- Use `Interface` and `Union` types for search queries that output multiple `Object` types

## Book Genres as an `Enum`

So far we've captured some essential metadata about books including their titles, authors, summaries, and cover image URLs, but it would be useful if we could categorize books by their genre too. To do that, we can add a `genre` field to the `Book` type and also include a `genre` field in the `CreateBookInput`. At first glance, it seems logical that the output type for the `genre` field would be a `String` `Scalar` corresponding to the genre name. However, Bibliotech has a small number of genres that may be applied to books, so we would also need some kind of runtime validation to ensure this requirement is enforced.

Luckily, the fourth named type that we will explore will allow us to do exactly that without manually adding runtime validation on the genre string when a new book is created. An *Enumeration* type, or `Enum` type for short, is like a `Scalar` type in a GraphQL schema but with the exception that it is limited to a finite set of defined values.<sup>1</sup>

To define a `Genre` `Enum` type, we use the `enum` keyword and list the allowed values. As a common convention, the values are defined in all caps:

---

<sup>1</sup><https://spec.graphql.org/June2018/#sec-Enums>

server/src/graphql/typeDefs.js

```
import { gql } from "apollo-server";

const typeDefs = gql`  
enum Genre {  
    ADVENTURE  
    CHILDREN  
    CLASSICS  
    COMIC_GRAPHIC_NOVEL  
    DETECTIVE_MYSTERY  
    DYSTOPIA  
    FANTASY  
    HORROR  
    HUMOR  
    NON_FICTION  
    SCIENCE_FICTION  
    ROMANCE  
    THRILLER  
    WESTERN  
}  
  
# ...  
;  
  
export default typeDefs;
```

We can now use the `Genre` type in the schema as we would an Object or Scalar type. Let's put it to use by adding a `genre` field to the `CreateBookInput` and `Book` types:

server/src/graphql/typeDefs.js

```
import { gql } from "apollo-server";

const typeDefs = gql`  
# ...  
  
input CreateBookInput {  
    authorIds: [ID]  
    cover: String  
    genre: Genre  
    title: String!  
}
```

```
# ...  
  
type Book {  
  id: ID!  
  authors: [Author]  
  cover: String  
  genre: Genre  
  reviews: [Review]  
  summary: String  
  title: String!  
}  
;  
  
export default typeDefs;
```

We'll also need to update the `createBook` method in the `JsonServerApi` data source to handle the genre value and save it to `db.json`:

`server/src/graphql/dataSources/JsonServerApi.js`

```
// ...  
  
class JsonServerApi extends RESTDataSource {  
  // ...  
  
  async createBook({ authorIds, cover, genre, summary, title }) {  
    const book = await this.post("/books", {  
      ...(cover && { cover }),  
      ...(genre && { genre }),  
      ...(summary && { summary }),  
      title  
    });  
  
    // ...  
    return book;  
  }  
  
  // ...  
}  
  
export default JsonServerApi;
```

Try creating a new book now to confirm that the genre is added to the book:

#### *GraphQL Mutation*

```
mutation CreateBookMutation($input: CreateBookInput!) {
  createBook(input: $input) {
    id
    authors {
      name
    }
    cover
    genre
  }
}
```

#### *Mutation Variables*

```
{
  "input": {
    "authorIds": ["1"],
    "cover": "http://covers.openlibrary.org/b/isbn/9781529034530-L.jpg",
    "genre": "SCIENCE_FICTION",
    "title": "The Restaurant at the End of the Universe"
  }
}
```

#### *API Response*

```
{
  "data": {
    "createBook": {
      "id": "3",
      "authors": [
        {
          "name": "Douglas Adams"
        }
      ],
      "cover": "http://covers.openlibrary.org/b/isbn/9781529034530-L.jpg",
      "genre": "SCIENCE_FICTION"
    }
  }
}
```

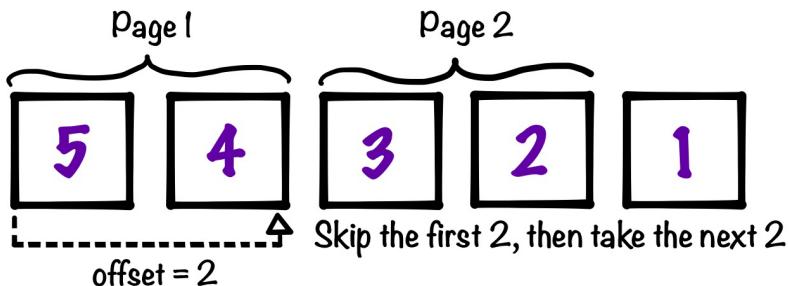
## Pagination Options for GraphQL APIs

Now that we've completed our Enum type warm-up round, we'll create some additional options to help sort data returned from certain list fields. Simultaneously, we'll add additional arguments to support paginating those fields where potentially long lists of data may be returned. But before we do that, we should explore what kinds of pagination options are commonly used with GraphQL APIs.

### Offset-Based

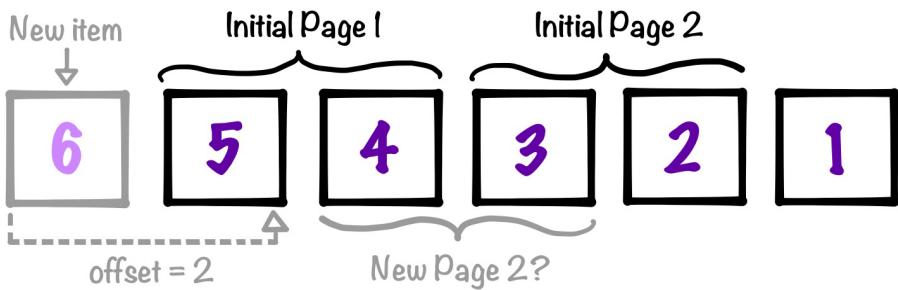
Historically, offset-based pagination has been a popular choice for paginating results from a database. With offset-based pagination, a client provides information about the number of results it wants to receive per page (called the *limit*) and how many results to skip before retrieving the limited number of items (called the *offset*). The server uses these criteria to query the database for that specific set of results (setting a default limit and offset, if necessary).

To visualize how offset-based pagination works, imagine you have a dataset with five items in it and you want to retrieve the second page of those items sorted in descending order with a limit of two items per page:



Offset-based pagination is useful when you need to know the total number of pages available. It can also easily support *bi-directional* pagination. Bi-directional pagination allows you to jump back and forth between pages or to navigate to a specific page within the results. This kind of navigation is often used on blogs and news sites.

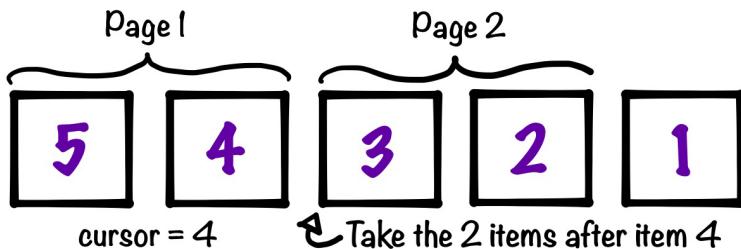
However, there can be performance downsides to this approach if the queried database has a lot of records in it. Further, if new records are added to the database at a high frequency, then the page window may become mismatched with real-time reality, resulting in duplicate or missed records in the pages of results. To illustrate this potential pitfall, imagine retrieving the first page of results from our dataset. While you're browsing those results, a new sixth item is added before requesting the second page. Suddenly, the paging window shifts back one position, and the fourth item will now confusingly appear at the end of the first page and the start of the second page:



### Cursor-Based

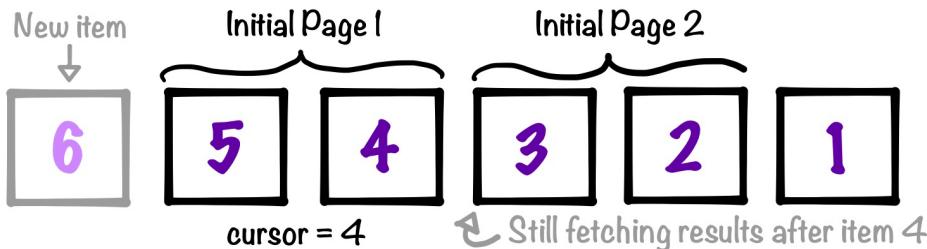
Cursor-based pagination uses (surprise!) a *cursor* to progress through results in a dataset. A cursor is a pointer to a specific result in a dataset and can be anything that makes sense to the back-end application as long as it's a unique, sequential value. As a client navigates through subsequent pages, the server returns results after the item marked by the cursor value.

Cursor-based pagination on our five-item dataset (once again in descending order with two items per page) can be visualized as follows:



The nature of the cursor itself is inconsequential to the client—the client just needs to send this value back to the server on subsequent requests so the server knows from which point it should retrieve more results.

Cursor-based pagination is well-suited to datasets updated at high velocities because it helps address the issue of page window inaccuracies that can happen with offset-based pagination. If a sixth item is added to our dataset after retrieving the first page, then there will be no confusion about where to start the second page when using a cursor:



This style of pagination does have a trade-off though. A cursor-based approach has the downside of not providing any way to jump to a specific page number or calculate the total number of pages reliably. However, if you're building an app that will be updated rapidly and with infinite scrolling implemented in the user interface to browse content, then the lack of numbered pages and total page counts likely won't be deal-breakers.

## Relay-Style

If you've previously researched pagination options for GraphQL APIs, you have likely encountered the pagination style used by [Relay](#). Relay-style pagination is an opinionated flavor of cursor-based pagination for GraphQL APIs. Relay itself is a JavaScript framework that can be used as a client to retrieve and cache data from a GraphQL API, just as Apollo Client does. It was created by Facebook and was designed with Facebook-level applications in mind (in other words, applications with lots of data in lists that are read and written at a high velocity).

Relay's barriers to entry are a bit higher than Apollo Client, so Relay itself often isn't the first package developers reach for when getting started with GraphQL. However, Relay offers a useful model for how to handle paginated data in GraphQL APIs in what it calls a [cursor connection specification](#).

An important thing to keep in mind with Relay-style pagination is that it is *uni-directional* by design. If you need to implement "Previous Page" and "Next Page" buttons to traverse content in an app, then Relay-style pagination probably won't work well for you (although a quick Google search will reveal some proposed workarounds for supporting bi-directional paging with Relay). However, if your user interface requires infinite scrolling to load additional pages of results, then this approach will be a good fit for you.

As we will see, Relay is very opinionated about how pagination requests are made via query arguments, how paginated types are structured, and also how the paginated query outputs data. A deep dive into Relay pagination internals is outside the scope of this book, but you can learn how to implement Relay-style pagination from scratch with MongoDB in [Advanced GraphQL with Apollo & React](#).

## So Which Do We Choose?

Before we choose a style of pagination and begin applying pagination arguments to a field, we have a very important question to answer first: *does this field need to be paginated in the first place?* For fields that will only ever return short lists of data, we can typically skip pagination and save client developers the hassle of wrangling pagination-related arguments for these fields.

Another important consideration when choosing how to paginate a field is to make sure that the style of pagination is appropriate for client use cases while also still technically feasible in regards to how the data is stored and may be retrieved from its database. Given that JSON Server has better support for offset-based pagination, we will keep things simple and choose this approach for paginating fields in this GraphQL API.

### Pagination Pro Tip:

It's not necessary to select a single style of pagination to use throughout an entire API—the pagination style should be driven by product requirements where possible. However, for the sake of the client developers' sanity, do ensure that pagination arguments are named and applied as consistently as possible across fields.

## Add Pagination for Authors, Books, Reviews, and Users

In our API's current state, there are seven fields that return lists of results. We will proceed with adding pagination arguments to the following five fields based on the likelihood of each of these fields potentially returning more data than we would want to deliver in a single response:

- On the `Query` type: `authors` and `books`
- On the `Book` type: `reviews`
- On the `User` type: `reviews` and `library`

That just leaves the `Book` type's `authors` field and the `Author` type's `books` field. We can be reasonably assured that most books will typically have at most a few authors, so we will leave the `authors` field as a basic list on the `Book` type. Whether to paginate the `books` field on the `Author` type is a less obvious choice—for the time being, we will leave this field as a basic list, but as time goes on, if an increasing number of particularly prolific authors are added to Bibliotech, then this field may need to be evolved to include pagination arguments.

To begin, we'll add some pagination-related Enum types to the schema that will allow a client to indicate how it would like the results ordered:

`server/src/graphql/typeDefs.js`

```
import { gql } from "apollo-server";
```

```
const typeDefs = gql`  
# ...  
  
enum AuthorOrderBy {  
  NAME_ASC  
  NAME_DESC  
}  
  
enum BookOrderBy {  
  TITLE_ASC  
  TITLE_DESC  
}  
  
enum LibraryOrderBy {  
  ADDED_ON_ASC  
  ADDED_ON_DESC  
}  
  
enum ReviewOrderBy {  
  REVIEWED_ON_ASC  
  REVIEWED_ON_DESC  
}  
  
# ...  
`;  
  
export default typeDefs;
```

We'll also add a `PageInfo` Object type that will be returned with paginated fields so the client knows what page it's on, whether there are previous and next pages, how many items were retrieved, and what the total count is:

`server/src/graphql/typeDefs.js`

```
import { gql } from "apollo-server";  
  
const typeDefs = gql`  
# ...  
  
type PageInfo {  
  hasNextPage: Boolean  
  hasPrevPage: Boolean
```

```
    page: Int
    perPage: Int
    totalCount: Int
}

# ...
`;

export default typeDefs;
```

Because we can only output a single type from a field, we'll need to wrap each list of paginated results and the PageInfo type in another Object type and return those types from the fields where pagination is used instead:

*server/src/graphql/typeDefs.js*

```
import { gql } from "apollo-server";

const typeDefs = gql` 
# ...

type Authors {
  results: [Author]
  pageInfo: PageInfo
}

# ...

type Books {
  results: [Book]
  pageInfo: PageInfo
}

# ...

type Reviews {
  results: [Review]
  pageInfo: PageInfo
}

# ...
`;
```

```
export default typeDefs;
```

Next, we'll update all of the fields that will be paginated to include arguments to indicate the number of items to fetch per page, the page number to fetch, and the sort order of the items:

*server/src/graphql/typeDefs.js*

```
import { gql } from "apollo-server";

const typeDefs = gql`  
  # ...  
  
  type Book {  
    id: ID!  
    authors: [Author]  
    cover: String  
    genre: Genre  
    reviews(limit: Int = 20, orderBy: ReviewOrderBy, page: Int): Reviews  
    summary: String  
    title: String!  
  }  
  
  # ...  
  
  type User {  
    id: ID!  
    email: String!  
    library(limit: Int = 20, orderBy: LibraryOrderBy, page: Int): Books  
    name: String  
    reviews(limit: Int = 20, orderBy: ReviewOrderBy, page: Int): Reviews  
    username: String!  
  }  
  
  # ...  
  
  type Query {  
    author(id: ID!): Author  
    authors(limit: Int = 20, orderBy: AuthorOrderBy, page: Int): Authors  
    book(id: ID!): Book  
    books(limit: Int = 20, orderBy: BookOrderBy, page: Int): Books  
    review(id: ID!): Review
```

```
    user(username: String!): User
  }
};

export default typeDefs;
```

Note in the code above that we have set a *default argument* for the `limit` argument of 20 results for each of the paginated fields. Default arguments are a handy way to communicate to API consumers if any behavior will take effect by default when no value is supplied for a nullable argument.

Now it's time to do some work in the `JsonServerApi` data source to support paginated results for these new and improved fields. We've already seen examples of how a query parameter can be used to filter results from the REST API, such as in the `getUser` method where we make a request to `/users?username=${username}` to get the user that matches a specific username. We can also use query parameters to assist with paginating and sorting requests from the REST API. For example, to fetch the second page of authors where 20 authors are listed per page, we would add the following query parameters:

```
/authors?_page=2&_limit=20
```

Similarly, to fetch authors in ascending order by name, we could use this query parameter:

```
/authors?_sort=name&_order=asc
```

Note that while JSON Server does have some notion of an offset query parameter (called `_start`), for the sake of convenience we will use the built-in `page` parameter instead so we don't have to manually calculate the starting point for each page. To parse the incoming field arguments into well-formatted query parameters, we're going to create a new method called `parseParams` in the `JsonServerApi` data source:

`server/src/graphql/dataSources/JsonServerApi.js`

```
// ...

class JsonServerApi extends RESTDataSource {
  // ...

  parseParams({ limit, orderBy, page, ...rest }) {
    if (limit && limit > 100) {
      throw new UserInputError("Maximum of 100 results per page");
    }
  }
}
```

```
const paginationParams = [];
paginationParams.push(`_limit=${limit}`, `_page=${page || "1"}`);

// Parse the `orderBy` argument into a `_sort` argument
// Handle other parameters collected in `rest`
// Return the full-assembled query string
}

// ...
}

export default JsonServerApi;
```

It's a good idea to set an upper limit on the number of results a client can fetch from a paginated field to prevent potential abuse of the API, so here we set a maximum of 100 results per request and throw an error if that limit is exceeded. If the requested number of items is fewer than 101, we create segments for the parts of the query string the will specify the `_limit` and `_page` parameters.

Assembling the string for the `_sort` parameter will be a bit more challenging. Recall that the Enum values we created are formatted in all caps with underscores used for whitespace:

```
NAME_ASC
```

But the query parameter must be formatted as follows:

```
_sort=name&_order=asc
```

Converting the `NAME_ASC` string to lowercase and splitting it at the underscore would work for this Enum value, but what about the `ADDED_ON_ASC` value from the `LibraryOrderBy` type (which corresponds to the `createdAt` field for a `userBooks` item)? Luckily, Apollo Server provides a solution that will help us avoid writing a lot of imperative code here. Typically, we expect to handle an Enum type's value as they are defined in the schema, but occasionally, we need a way to map the values that are defined in a schema to different *internal values* for use inside an application. While the values of the Enum type will be maintained in the public-facing API, the internal values will be available inside of any resolvers functions that accept the Enum as an argument.<sup>2</sup>

To use this feature, we'll need to add maps of the Enum type's values to the internal values in the resolvers that we pass to Apollo Server:

<sup>2</sup><https://www.apollographql.com/docs/apollo-server/schema/schema/#internal-values-advanced>

server/src/graphql/resolvers.js

```
const resolvers = {
  AuthorOrderBy: {
    NAME_ASC: "name_asc",
    NAME_DESC: "name_desc"
  },
  BookOrderBy: {
    TITLE_ASC: "title_asc",
    TITLE_DESC: "title_desc"
  },
  LibraryOrderBy: {
    ADDED_ON_ASC: "createdAt_asc",
    ADDED_ON_DESC: "createdAt_desc"
  },
  ReviewOrderBy: {
    REVIEWED_ON_ASC: "createdAt_asc",
    REVIEWED_ON_DESC: "createdAt_desc"
  },
  // ...
}

export default resolvers;
```

When we set the default value for the `limit` argument previously, you may have wondered why we didn't do the same for the `orderBy` arguments. Ideally, we would have, but there is an issue with v4.x of the GraphQL Tools package (an Apollo Server dependency) that prevents Enum type values from being mapped to their internal values when passed as default arguments.

You can view the following GitHub issue for more details:

<https://github.com/ardatan/graphql-tools/issues/715>

With this code in place, any time we reference an argument in a resolver function that uses one of these Enum values, the internal value will be conveniently available to use instead. Now we can parse the `_sort` and `_order` parameters in the `parseParams` method:

server/src/graphql/dataSources/JsonServerApi.js

```
// ...
```

```
class JsonServerApi extends RESTDataSource {  
    // ...  
  
    parseParams({ limit, orderBy, page, ...rest }) {  
        // ...  
  
        const [sort, order] = orderBy ? orderBy.split("_") : [];  
        // Handle other parameters collected in `rest`  
        // Return the full-assembled query string  
    }  
  
    // ...  
}  
  
export default JsonServerApi;
```

Lastly, we'll handle any other query parameters that were passed into `parseParams`, join them using the `&` character, and return the resulting string:

`server/src/graphql/dataSources/JsonServerApi.js`

```
// ...  
  
class JsonServerApi extends RESTDataSource {  
    // ...  
  
    parseParams({ limit, orderBy, page, ...rest }) {  
        // ...  
  
        const [sort, order] = orderBy ? orderBy.split("_") : [];  
        const otherParams = Object.keys(rest).map(key =>  
            `${key}=${rest[key]}`  
        );  
        const queryString = [  
            ...(sort ? `_sort=${sort}` : []),  
            ...(order ? `_order=${order}` : []),  
            ...paginationParams,  
            ...otherParams  
        ].join("&");  
  
        return queryString ? `?${queryString}` : "";  
    }  
}
```

```
// ...
}

export default JsonServerApi;
```

Before we update the data source methods for the paginated fields, we'll also need a helper to parse the pagination-related metadata into an object that can supply the data for the new `PageInfo` type in the schema. To do that, we'll need to know what `limit` and `page` arguments were used to paginate the results, and we must also use the `X-Total-Count` header in the REST API response to supply the data for the `totalCount` field. Additionally, we'll need to use the `Link` header from the response to determine if there are previous and next pages. We'll begin by installing a package in `server` that helps parse the `Link` header into an object:

```
npm i parse-link-header@1.0.1
```

Next, we'll override the `didReceiveResponse` method of the parent `RESTDataSource` class to intercept the `Link` and `X-Total-Count` headers before parsing the response or returning an error:

`server/src/graphql/dataSources/JsonServerApi.js`

```
// ...

class JsonServerApi extends RESTDataSource {
  baseURL = process.env.REST_API_BASE_URL;

  async didReceiveResponse(response) {
    if (response.ok) {
      this.linkHeader = response.headers.get("Link");
      this.totalCountHeader = response.headers.get("X-Total-Count");
      return this.parseBody(response);
    } else {
      throw await this.errorFromResponse(response);
    }
  }

  // ...
}

export default JsonServerApi;
```

Now we can use the `limit` and `page` arguments, the `totalCount` and `linkHeader` properties we just set, and the `parseLinkHeader` function to populate the fields for the `PageInfo` type:

`server/src/graphql/dataSources/JsonServerApi.js`

```
// ...
import parseLinkHeader from "parse-link-header";

class JsonServerApi extends RESTDataSource {
// ...

parsePageInfo({ limit, page }) {
  if (this.totalCountHeader) {
    let hasNextPage, hasPrevPage;

    if (this.linkHeader) {
      const { next, prev } = parseLinkHeader(this.linkHeader);
      hasNextPage = !!next;
      hasPrevPage = !!prev;
    }

    return {
      hasNextPage: hasNextPage || false,
      hasPrevPage: hasPrevPage || false,
      page: page || 1,
      perPage: limit,
      totalCount: this.totalCountHeader
    };
  }

  return null;
}

// ...
}

export default JsonServerApi;
```

At long last, we can finally update the `getAuthors`, `getBookReviews`, `getBooks`, and `getUserLibrary` and `getUserReviews` methods. We'll start with `getAuthors`:

*server/src/graphql/dataSources/JsonServerApi.js*

```
// ...

class JsonServerApi extends RESTDataSource {
    // ...

    async getAuthors({ limit, page, orderBy = "name_asc" }) {
        const queryString = this.parseParams({
            ...(limit && { limit }),
            ...(page && { page }),
            orderBy
        });
        const authors = await this.get(`/authors${queryString}`);
        const pageInfo = this.parsePageInfo({ limit, page });

        return { results: authors, pageInfo };
    }

    // ...
}

export default JsonServerApi;
```

In the revised code above, we now mark the `getAuthors` methods as `async` so we can use the `await` keyword inside of it. We first construct the string of query parameters using the `parseParams` method, then fetch the results from the `/authors` endpoint with the string appended, parse the page metadata with the `parsePageInfo` method, and finally return an object containing the `results` and `pageInfo` properties, which correspond to the fields in the new `Authors` Object type.

Now we'll follow a similar pattern to update the methods for the rest of the paginated fields:

*server/src/graphql/dataSources/JsonServerApi.js*

```
// ...

class JsonServerApi extends RESTDataSource {
    // ...

    async getBookReviews(bookId, { limit, page, orderBy = "createdAt_desc" })
    {
        const queryString = this.parseParams({
```

```
    ...({limit} && { limit }),
    ...({page} && { page }),
    bookId,
    orderBy
  });
const reviews = await this.get(`/reviews${queryString}`);
const pageInfo = this.parsePageInfo({ limit, page });

  return { results: reviews, pageInfo };
}

async getBooks({ limit, page, orderBy = "title_asc" }) {
  const queryString = this.parseParams({
    ...({limit} && { limit }),
    ...({page} && { page }),
    orderBy
  });
  const books = await this.get(`/books${queryString}`);
  const pageInfo = this.parsePageInfo({ limit, page });

  return { results: books, pageInfo };
}

// ...

async getUserLibrary(userId, { limit, page, orderBy = "createdAt_desc" }) {
  const queryString = this.parseParams({
    _expand: "book",
    ...({limit} && { limit }),
    ...({page} && { page }),
    orderBy,
    userId
  });
  const items = await this.get(`/userBooks${queryString}`);
  const books = items.map(item => item.book);
  const pageInfo = this.parsePageInfo({ limit, page });

  return { results: books, pageInfo };
}
```

```
async getUserReviews(userId, { limit, page, orderBy = "createdAt_desc" }) {
  const queryString = this.parseParams({
    ...(limit && { limit }),
    ...(page && { page }),
    orderBy,
    userId
  });
  const reviews = await this.get(`/reviews${queryString}`);
  const pageInfo = this.parsePageInfo({ limit, page });

  return { results: reviews, pageInfo };
}

// ...
}

export default JsonServerApi;
```

Note that we can no longer use the aliased /userBooks route in the `getUserLibrary` method because aliased routes can't use query strings in JSON Server. As a result, we have to manually expand the related book records for each item in the response.

Before testing out the paginated fields, we must update the resolver function for each field to pass through the pagination arguments to the data source method that it calls:

`server/src/graphql/resolvers.js`

```
const resolvers = {
  // ...
  Book: {
    // ...
    reviews(book, args, { dataSources }, info) {
      return dataSources.jsonServerApi.getBookReviews(book.id, args);
    }
  },
  // ...
  User: {
    library(user, args, { dataSources }, info) {
      return dataSources.jsonServerApi.getUserLibrary(user.id, args);
    },
    reviews(user, args, { dataSources }, info) {
      return dataSources.jsonServerApi.getUserReviews(user.id, args);
    }
}
```

```
        },
    },
  Query: {
    // ...
    authors(root, args, { dataSources }, info) {
      return dataSources.jsonServerApi.getAuthors(args);
    },
    // ...
    books(root, args, { dataSources }, info) {
      return dataSources.jsonServerApi.getBooks(args);
    },
    // ...
  },
  // ...
}

export default resolvers;
```

Now let's try fetching a paginated list of books. If you only have one book stored in db.json right now, then run the `createBook` mutation a few times with some new book data first so you can ensure the pagination is working as expected:

#### GraphQL Query

```
query GetBooks($limit: Int, $orderBy: BookOrderBy, $page: Int) {
  books(limit: $limit, orderBy: $orderBy, page: $page) {
    results {
      title
    }
    pageInfo {
      hasNextPage
      hasPrevPage
      page
      perPage
      totalCount
    }
  }
}
```

*Query Variables*

```
{  
  "limit": 2,  
  "orderBy": "TITLE_DESC",  
  "page": 1  
}
```

*API Response*

```
{  
  "data": {  
    "books": {  
      "results": [  
        {  
          "title": "The Restaurant at the End of the Universe"  
        },  
        {  
          "title": "The Hitchhiker's Guide to the Galaxy"  
        }  
      ],  
      "pageInfo": {  
        "hasNextPage": true,  
        "hasPrevPage": false,  
        "page": 1,  
        "perPage": 2,  
        "totalCount": 3  
      }  
    }  
  }  
}
```

We can even try fetching a paginated list of books with a paginated list of reviews for each book:

*GraphQL Query*

```
query GetBooksWithReviews(  
  $booksLimit: Int  
  $booksOrderBy: BookOrderBy  
  $booksPage: Int  
  $reviewsLimit: Int  
  $reviewsOrderBy: ReviewOrderBy  
  $reviewsPage: Int
```

```
) {  
  books(limit: $booksLimit, orderBy: $booksOrderBy, page: $booksPage) {  
    results {  
      title  
      reviews(  
        limit: $reviewsLimit  
        orderBy: $reviewsOrderBy  
        page: $reviewsPage  
      ) {  
        results {  
          reviewedOn  
        }  
      }  
    }  
  }  
}
```

#### Query Variables

```
{  
  "booksLimit": 2,  
  "booksOrderBy": "TITLE_ASC",  
  "booksPage": 1,  
  "reviewsLimit": 1,  
  "reviewsOrderBy": "REVIEWED_ON_DESC",  
  "reviewsPage": 1  
}
```

#### API Response

```
{  
  "data": {  
    "books": {  
      "results": [  
        {  
          "title": "The Dispossessed",  
          "reviews": {  
            "results": []  
          }  
        },  
        {  
          "title": "The Hitchhiker's Guide to the Galaxy",  
        }  
      ]  
    }  
  }  
}
```

```
    "reviews": {
      "results": [
        {
          "reviewedOn": "2021-02-20T15:02:28.308Z"
        }
      ]
    }
  }
}
```

Before moving on, try rerunning the `authors` query and the `user` query with paginated `library` and `review` fields to confirm that they work as expected.

## Interfaces and Unions: What Are They Good For?

Pagination is finally functional and we can now make further use of the `parseParams` method to power two new search queries for the Bibliotech API. The first query will allow users to search for books by author name or book title. The second query will support searching for people by name, including both authors and users in the results. But we have been limited to setting a single output type for each field so far, so how will we return authors and books from the first query and authors and users from the second?

There are two ways we can do this with GraphQL. So far we have used Scalar, Object, Input, and Enum types in our schema, but there are still two named types we have yet to explore—one is the *Interface* type and the other is the *Union* type. In a GraphQL schema, an Interface type is an abstract type that specifies fields that must be included by any other concrete types that implement it. In other words, it guarantees types include a common set of fields. For example:

```
interface Plant {
  species: String
}

type Flower implements Plant {
  species: String
  petalColor: [String]
}

type Tree implements Plant {
  species: String
```

```
    deciduous: Boolean
}

type Query {
  plants: [Plant]
}
```

In this example, because the `Flower` and `Tree` types implement the `Plant` Interface, they must have a `species` field (in addition to whatever type-specific fields they may support). When we run the `plants` query, we can expect to get back results containing one or both types. This shared behavior of flowers and trees (where both are identifiable as plant species and may be returned by the `plants` query) is a good indication that an Interface type is warranted here—we want our Interface types to serve a functional purpose in the API and not simply define a contract of shared fields.

A Union type, on the other hand, is an abstract type like the Interface type but it doesn't require any common fields between the types. That means we use Union types when we simply need a field to return any number of unrelated Object types. Reworking our plants example to use a Union would look like this:

```
union Plant = Flower | Tree

type Flower {
  petalColor: [String]
}

type Tree {
  species: String
  deciduous: Boolean
}

type Query {
  plants: [Plant]
}
```

Here, there is no guarantee that `Flower` and `Tree` will both contain `species` field, but as with the Interface type, we can still have a `plants` query that outputs either Object type. To implement our search queries, we will use a `Person` Interface type for the query that searches for authors and users (given that they are conceptually similar and also share `id` and `name` fields) and we'll use a Union type for the query that search for books by author name or book title (given that authors and books are two very different kinds of resources).

## Add a Query to Search for People by Name

To set up the `searchPeople` query, we'll first create an Enum type for ordering the results and the `Person` Interface containing non-null `id` and `name` fields:

`server/src/graphql/typeDefs.js`

```
import { gql } from "apollo-server";

const typeDefs = gql`  
  # ...  
  
  enum SearchOrderBy {  
    RESULT_ASC  
    RESULT_DESC  
  }  
  
  interface Person {  
    id: ID!  
    name: String!  
  }  
  
  # ...  
`;  
  
export default typeDefs;
```

Next, we'll update the `Author` and `User` Object types to implement `Person`:

`server/src/graphql/typeDefs.js`

```
import { gql } from "apollo-server";

const typeDefs = gql`  
  # ...  
  
  type Author implements Person {  
    # ...  
  }  
  
  # ...  
  
  type User implements Person {
```

```
    # ...
}

# ...
`;

export default typeDefs;
```

Now we'll add the `searchPeople` query. We'll add an `exact` argument to the field to indicate if we should search for an exact match on the name value or use a full-text search instead:

`server/src/graphql/typeDefs.js`

```
import { gql } from "apollo-server";

const typeDefs = gql`  
# ...

type Query {  
# ...  
  searchPeople(  
    exact: Boolean = false  
    orderBy: SearchOrderBy  
    query: String!  
  ): [Person]  
  user(username: String!): User  
}  
  
# ...  
`;  
  
export default typeDefs;
```

Due to limitations with the mocked REST API, we will not be able to paginate search results reliably. We will instead return a maximum of 50 results each of author and user objects that are sorted alphabetically by name to simulate a search query. However, if we were using a backing data store that did support full-text search across different data types, then we would typically add pagination options to a field such as this given its potential to return a long list of matching results. We would also likely return results by relevance if this sorting option is available.

Back in the `JsonServerApi` data source, we'll add a `searchPeople` method to fetch matching authors and users using the `name_like` query parameter in the request to the REST API and then sort them by name:

`server/src/graphql/dataSources/JsonServerApi.js`

```
// ...

class JsonServerApi extends RESTDataSource {
    // ...

    async searchPeople({ exact, query, orderBy = "RESULT_ASC" }) {
        const queryString = this.parseParams({
            ...(exact ? { name: query } : { q: query }),
            limit: 50
        });
        const authors = await this.get(`authors${queryString}`);
        const users = await this.get(`users${queryString}`);
        const results = []
            .concat(authors, users)
            .sort((a, b) =>
                orderBy === "RESULT_ASC"
                    ? a.name.localeCompare(b.name)
                    : b.name.localeCompare(a.name)
            );

        return results;
    }

    // ...
}

export default JsonServerApi;
```

Now we'll use that method in a new resolver for the `searchPeople` field on the `Query` type:

`server/src/graphql/resolvers.js`

```
const resolvers = {
    // ...
    Query: {
        // ...
        searchPeople(root, args, { dataSources }, info) {
```

```
        return dataSources.jsonServerApi.searchPeople(args);
    },
    // ...
},
// ...
}

export default resolvers;
```

While it may seem like we're done at this point, we still have one more addition to make to the resolvers. When an abstract type is added to the schema, we must also provide a special `__resolveType` resolver so Apollo Server understands what kind of Object type is being returned from the field. To implement a `__resolveType` resolver for the `Person` type, we can use the presence of a `username` field on the object to identify if it represents a user or an author. We then return the name of the corresponding Object type as a string from the resolver:

`server/src/graphql/resolvers.js`

```
const resolvers = {
// ...
Person: {
  __resolveType(obj, context, info) {
    if (obj.username) {
      return "User";
    } else {
      return "Author";
    }
  }
},
// ...
}

export default resolvers;
```

Now we can try out our search first search query (you may want to add some authors and users with similar names first so that the query returns both kinds of objects). When we make a query that outputs an Interface type, we add any of the common fields as selections as usual, but we must use an *inline fragment* to fetch fields that are specific to any type as follows:

*GraphQL Query*

```
query SearchPeople($query: String!, $orderBy: SearchOrderBy) {
  searchPeople(query: $query, orderBy: $orderBy) {
    id
    name
    ... on Author {
      books {
        title
      }
    }
    ... on User {
      username
    }
  }
}
```

*Query Variables*

```
{
  "query": "Alice",
  "orderBy": "RESULT_ASC"
}
```

*API Response*

```
{
  "data": {
    "searchPeople": [
      {
        "id": "1",
        "name": "Alice Liddell",
        "username": "rabbithole84"
      },
      {
        "id": "3",
        "name": "Alice Walker",
        "books": [
          {
            "title": "The Color Purple"
          }
        ]
      }
    ]
  }
}
```

```
        }
    ]
}
}
```

## Add a Query to Search Books by Author or Title

Our final task for this chapter is to add the search query that retrieves books by their titles or author names. To do that, we'll add a Union type definition for book search results and a `searchBooks` field on `Query` that returns the new type:

`server/src/graphql/typeDefs.js`

```
import { gql } from "apollo-server";

const typeDefs = gql`  
# ...

union BookResult = Book | Author

# ...

type Query {  
# ...  
  searchBooks(  
    exact: Boolean = false  
    orderBy: SearchOrderBy  
    query: String!  
  ): [BookResult]  
# ...  
}  
  
# ...  
`;  
  
export default typeDefs;
```

Next, we'll add a method to the `JsonServerApi` data source called `searchBooks` that will fetch matches from both the `/authors` and `/books` endpoints, and then combine and sort them:

*server/src/graphql/dataSources/JsonServerApi.js*

```
// ...

class JsonServerApi extends RESTDataSource {
    // ...

    async searchBooks({ exact, query, orderBy = "RESULT_ASC" }) {
        const bookQueryString = this.parseParams({
            ...(exact ? { title: query } : { q: query }),
            limit: 50
        });
        const authorQueryString = this.parseParams({
            ...(exact ? { name: query } : { q: query }),
            limit: 50
        });

        const authors = await this.get(`/authors${authorQueryString}`);
        const books = await this.get(`/books${bookQueryString}`);
        const results = [].concat(authors, books).sort((a, b) => {
            const aKey = a.hasOwnProperty("title") ? "title" : "name";
            const bKey = b.hasOwnProperty("title") ? "title" : "name";

            return orderBy === "RESULT_ASC"
                ? a[aKey].localeCompare(b[bKey])
                : b[bKey].localeCompare(a[aKey]);
        });

        return results;
    }

    // ...
}

export default JsonServerApi;
```

Lastly, we'll update the resolvers. Just as with the previous Person Interface type, we must add a `__resolveType` resolver so Apollo Server can identify what kind of Object type is returned:

*server/src/graphql/resolvers.js*

```
const resolvers = {
    // ...
```

```
BookResult: {
  __resolveType(obj, context, info) {
    if (obj.title) {
      return "Book";
    } else {
      return "Author";
    }
  }
},
// ...
Query: {
  // ...
  searchBooks(root, args, { dataSources }, info) {
    return dataSources.jsonServerApi.searchBooks(args);
  },
  // ...
},
// ...
}

export default resolvers;
```

Again, just as with queries for Interface types, we then use inline fragments to select fields for type. You may wish to add authors and book titles with similar names to db.json before testing this query:

#### GraphQL Query

```
query SearchBooks($query: String!, $orderBy: SearchOrderBy) {
  searchBooks(query: $query, orderBy: $orderBy) {
    ... on Author {
      name
      books {
        title
      }
    }
    ... on Book {
      title
      authors {
        name
      }
    }
  }
}
```

```
    }  
}
```

### Query Variables

```
{  
  "query": "Adams",  
  "orderBy": "RESULT_ASC"  
}
```

### API Response

```
{  
  "data": {  
    "searchBooks": [  
      {  
        "name": "Douglas Adams",  
        "books": [  
          {  
            "title": "The Hitchhiker's Guide to the Galaxy"  
          },  
          {  
            "title": "The Restaurant at the End of the Universe"  
          }  
        ]  
      },  
      {  
        "title": "John Adams",  
        "authors": [  
          {  
            "name": "David McCullough"  
          }  
        ]  
      }  
    ]  
  }  
}
```

## Summary

In this chapter, we implemented some advanced GraphQL features in our schema. We first learned how to paginate results returned by fields with the support of Enum types as arguments. We also used the GraphQL types system’s two abstract types—Interfaces and Unions—to return results from fields that consist of multiple Object types.

In the next chapter, we’ll add further enhancements to the Bibliotech API using custom Scalar types and directives, and we’ll also provide documentation for the fields and types in the API.

## Chapter 5

# Documentation, Custom Scalars, and Custom Directives

In this chapter, we will:

- Document the type and fields in a GraphQL schema
- Add custom `DateTime` and `Rating` Scalar types and use them in the schema
- Add a custom schema directive to enforce unique values for fields

## Best Practice: Document the Schema

Throughout this chapter, we're going to learn how we can enhance the overall expressiveness of the substantial schema we've already assembled. Our first step will be adding documentation to the types and fields in the schema. According to the current specification, documentation is a first-class feature of GraphQL type systems.<sup>1</sup> To that end, GraphQL makes it possible to document the types directly in our schema alongside the type definitions themselves using *descriptions*. A description is written using Markdown syntax, defined in *BlockString* format, and placed immediately before the type, field, or argument that it describes.

The specification recommends that all GraphQL types, fields, and arguments provide a description unless it's completely self-evident what they are. Before adding anything else to our schema, it would be a good idea to document what we've already created so other developers will understand what's under the hood.

The *BlockString* descriptions can be formatted as follows:

```
"""
This is a
```

<sup>1</sup><https://spec.graphql.org/June2018/#sec-Descriptions>

```
multiline
description.
"""

"This is a single-line description."
```

We'll update all of our API's type definitions in `typeDefs.js` to include this style of documentation now. Once we've finished adding descriptions to the various types and fields, we'll be able to view them in the user interface of a GraphQL IDE such as Explorer too. First, we'll add a description for the `Genre` Enum type:

`server/src/graphql/typeDefs.js`

```
import { gql } from "apollo-server";

const typeDefs = gql` 
"""
Literary genres that classify books.
"""

enum Genre {
  # ...
}

# ...
`;

export default typeDefs;
```

If we wanted to, we could document each value in the `Enum` with a description as well, but the genres are relatively self-explanatory so we'll skip this for now. Next, we'll document the rest of the `Enum` types:

`server/src/graphql/typeDefs.js`

```
import { gql } from "apollo-server";

const typeDefs = gql` 
# ...

"""

Sorting options for authors.
"""
```

```
enum AuthorOrderBy {
  NAME_ASC
  NAME_DESC
}

"""
Sorting options for books.
"""

enum BookOrderBy {
  TITLE_ASC
  TITLE_DESC
}

"""
Sorting options for books in a user's library.
"""

enum LibraryOrderBy {
  ADDED_ON_ASC
  ADDED_ON_DESC
}

"""
Sorting options for reviews.
"""

enum ReviewOrderBy {
  REVIEWED_ON_ASC
  REVIEWED_ON_DESC
}

"""
Sorting options for search results.
"""

enum SearchOrderBy {
  RESULT_ASC
  RESULT_DESC
}

# ...
`;

export default typeDefs;
```

Now we'll document the abstract types:

server/src/graphql/typeDefs.js

```
import { gql } from "apollo-server";

const typeDefs = gql`  
  # ...  
  
  """  
    Specifies fields shared by people (including authors and users).  
  """  
  interface Person {  
    "The unique ID of the person."  
    id: ID!  
    "The full name of the person."  
    name: String!  
  }  
  
  """  
    Types that can be returned from book-related search results.  
  """  
  union BookResult = Book | Author  
  
  # ...  
`;  
  
export default typeDefs;
```

And all of the Object types:

server/src/graphql/typeDefs.js

```
import { gql } from "apollo-server";

const typeDefs = gql`  
  # ...  
  
  """  
    An author is a person who wrote one or more books.  
  """  
  type Author implements Person {  
    "The unique ID of the author."  
    id: ID!  
    "Books that were authored or co-authored by this person."  
  }  
`;
```

```
books: [Book]
  "The full name of the author."
  name: String!
}

"""
A list of author results with pagination information.
"""

type Authors {
  "A list of author results."
  results: [Author]
  "Information to assist with pagination."
  pageInfo: PageInfo
}

"""
A written work that can be attributed to one or more authors and can be
reviewed by users.
"""

type Book {
  "The unique ID of the book."
  id: ID!
  "The author(s) who wrote the books"
  authors: [Author]
  "The URL of the book's cover image."
  cover: String
  "A literary genre to which the book can be assigned."
  genre: Genre
}

"""
User-submitted reviews of the book.

Default sort order is REVIEWED_ON_DESC.
"""

reviews(limit: Int = 20, orderBy: ReviewOrderBy, page: Int): Reviews
  "A brief description of the book's content."
  summary: String
  "The title of the book."
  title: String!
}

"""
A list of book results with pagination information.
"""
```

```
"""
type Books {
    "A list of book results."
    results: [Book]
    "Information to assist with pagination."
    pageInfo: PageInfo
}

"""
Contains information about the current page of results.
"""

type PageInfo {
    "Whether there are items to retrieve on a subsequent page."
    hasNextPage: Boolean
    "Whether there are items to retrieve on a preceding page."
    hasPrevPage: Boolean
    "The current page number."
    page: Int
    "The number of items retrieved per page."
   perPage: Int
    "The total item count across all pages."
    totalCount: Int
}

"""
A user-submitted assessment of a book that may include a numerical
rating.
"""

type Review {
    "The unique ID of the review."
    id: ID!
    "The book to which the review applies."
    book: Book
    "The user's integer-based rating of the book (from 1 to 5)."
    rating: Int!
    "The date and time the review was created."
    reviewedOn: String
    "The user who submitted the book review."
    reviewer: User!
    "The text-based content of the review."
    text: String
}
```

```
"""
A list of review results with pagination information.
"""

type Reviews {
    "A list of review results."
    results: [Review]
    "Information to assist with pagination."
    pageInfo: PageInfo
}

"""
A user account provides authentication and library details.
"""

type User implements Person {
    "The unique ID of the user."
    id: ID!
    "The email address of the user (must be unique)."
    email: String!
}

"""
A list of books the user has added to their library.

Default sort order is ADDED_ON_DESC.
"""

library(limit: Int = 20, orderBy: LibraryOrderBy, page: Int): Books
    "The full name of the user."
    name: String!
    """

    A list of book reviews created by the user.

    Default sort order is REVIEWED_ON_DESC.
    """

reviews(limit: Int = 20, orderBy: ReviewOrderBy, page: Int): Reviews
    "The user's chosen username (must be unique)."
    username: String!
}

# ...
`;

export default typeDefs;
```

In the descriptions for the paginated fields belonging to `Book` and `User` above, we indicate what the default sort order is to compensate partially for the limitation on using `Enum` types with internal values as default arguments for fields (see Chapter 4 for further explanation). Next, we'll add documentation for the Input Object types:

`server/src/graphql/typeDefs.js`

```
import { gql } from "apollo-server";

const typeDefs = gql`  
  # ...  
  
  """  
  Provides data to create a book.  
  """  
  input CreateBookInput {  
    "The IDs of the authors who wrote the book."  
    authorIds: [ID]  
    """  
      The URL of the book's cover image. Covers available via the Open  
      Library Covers API:  
  
      https://openlibrary.org/dev/docs/api/covers  
    """  
    cover: String  
    "A literary genre to which the book can be assigned."  
    genre: Genre  
    "A short summary of the book's content."  
    summary: String  
    "The title of the book."  
    title: String!  
  }  
  
  """  
  Provides data to create a review.  
  """  
  input CreateReviewInput {  
    "The unique ID of the book a user is reviewing."  
    bookId: ID!  
    "The user's integer-based rating of the book (from 1 to 5)."  
    rating: Int!  
    "The ID of the user submitting the review."  
    reviewerId: ID!
```

```
    "The text-based content of the review."
    text: String
}

"""

Provides data to create a user.
"""

input SignUpInput {
    "The email address of the user (must be unique)."
    email: String!
    "The full name of the user."
    name: String!
    "The user's chosen username (must be unique)."
    username: String!
}

"""

Provides data to add or remove books from a user's library.
"""

input UpdateLibraryBooksInput {
    "The IDs of the books to add or remove from the user's library."
    bookIds: [ID!]!
    "The ID of the user whose library should be updated."
    userId: ID!
}

"""

Provides data to update a review.
"""

input UpdateReviewInput {
    "The unique ID of the review a user is updating."
    id: ID!
    "The user's integer-based rating of the book (from 1 to 5)."
    rating: Int!
    "The text-based content of the review."
    text: String
}

# ...
;

export default typeDefs;
```

Now we'll document all of the fields on the root Query type:

*server/src/graphql/typeDefs.js*

```
import { gql } from "apollo-server";

const typeDefs = gql`  
  # ...  
  
  type Query {  
    "Retrieves a single author by ID."  
    author(id: ID!): Author  
    """  
      Retrieves a list of authors with pagination information.  
  
      Default sort order is NAME_ASC.  
    """  
    authors(limit: Int = 20, orderBy: AuthorOrderBy, page: Int): Authors  
    "Retrieves a single book by ID."  
    book(id: ID!): Book  
    """  
      Retrieves a list of books with pagination information.  
  
      Default sort order is TITLE_ASC.  
    """  
    books(limit: Int = 20, orderBy: BookOrderBy, page: Int): Books  
    "Retrieves a single book by ID."  
    review(id: ID!): Review  
    """  
      Performs a search of book titles and author names.  
  
      Default sort order is RESULTS_ASC.  
    """  
    searchBooks(  
      exact: Boolean = false  
      orderBy: SearchOrderBy  
      query: String!  
    ): [BookResult]  
    """  
      Performs a search of author and user names.  
  
      Default sort order is RESULTS_ASC.  
    """
```

```
searchPeople(
  exact: Boolean = false
  orderBy: SearchOrderBy
  query: String!
): [Person]
  "Retrieves a single user by username."
  user(username: String!): User
}

# ...
`;

export default typeDefs;
```

And lastly, the root `Mutation` type:

`server/src/graphql/typeDefs.js`

```
import { gql } from "apollo-server";

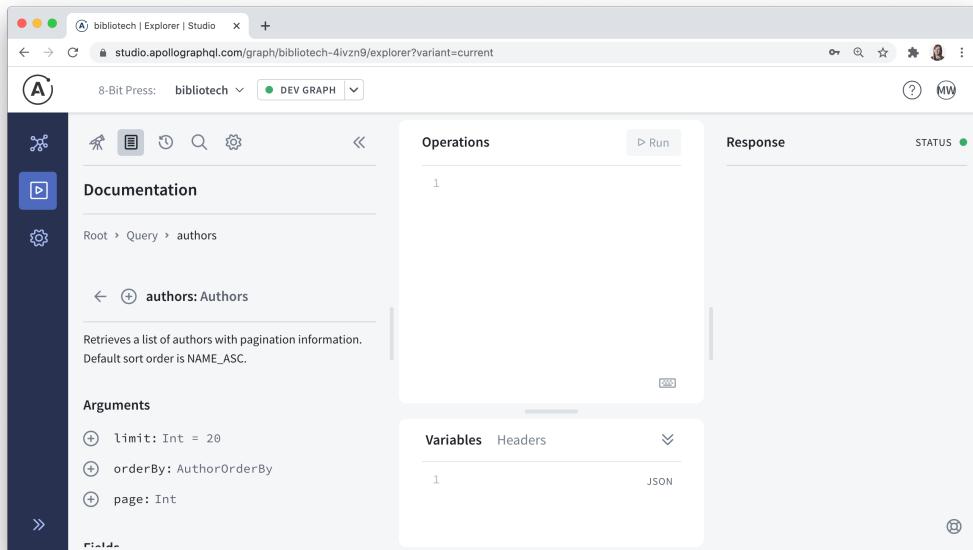
const typeDefs = gql` 
  # ...

  type Mutation {
    "Adds books to user's library."
    addBooksToLibrary(input: UpdateLibraryBooksInput!): User!
    "Creates a new author."
    createAuthor(name: String!): Author!
    "Creates a new book."
    createBook(input: CreateBookInput!): Book!
    "Creates a new review."
    createReview(input: CreateReviewInput!): Review!
    "Deletes a review."
    deleteReview(id: ID!): ID!
    "Remove books currently in a user's library."
    removeBooksFromLibrary(input: UpdateLibraryBooksInput!): User!
    "Creates a new user."
    signUp(input: SignUpInput!): User!
    "Updates a review."
    updateReview(input: UpdateReviewInput!): Review!
  }
`
```

```
# ...
;

export default typeDefs;
```

After saving the updated `typeDefs.js` file, the schema documentation will be visible in Explorer:



## Add a Custom DateTime Scalar

So far, we've put GraphQL's built-in `Int`, `String`, `Boolean`, and `ID` Scalar types to good use, but occasionally it would be useful to have a more purpose-built Scalar type at hand to use as an output type for a field. For instance, while a `String` type certainly works as an output type for the `Review` type's `reviewedOn` field, it would be helpful if there was further validation wired directly into this type to ensure that the string value is in a valid ISO 8601 format. Luckily, we can create a custom Scalar type for our GraphQL schema that we'll define as `DateTime`. The `DateTime` type will provide an extra measure of certainty for client applications that they will receive a date string in this precise format. It may also be used to validate argument values submitted with operations from the client.

The process for adding a custom Scalar type mirrors the workflow that we've already seen—that is, we will declare a type in the schema and then add a resolver so that GraphQL knows what to

do with that type. To begin, we'll add the following line to the top of the `typeDefs.js` file and update the `reviewedOn` field of the `Review` type to use the new Scalar type:

`server/src/graphql/typeDefs.js`

```
import { gql } from "apollo-server";

const typeDefs = gql`

  """
  An ISO 8601-encoded UTC date string.
  """

  scalar DateTime

  # ...

  """
  A user-submitted assessment of a book that may include a numerical
  rating.
  """

  type Review {
    # ...
    "The date and time the review was created."
    reviewedOn: DateTime!
    # ...
  }
`;

export default typeDefs;
```

Next, we need to define the date string-validating behavior of the `DateTime` type by instantiating a `GraphQLScalarType` from the `graphql` package. We'll organize this code in a `DateTimeType.js` file in a new subdirectory called `scalars` in `server/src/graphql`. We'll also need to install another package in `server` to assist with date string validation:

```
| npm i validator@13.5.2
```

This package is a handy library for doing string validation and sanitization in JavaScript. You'll need to import it along with the generic `ApolloError` from `apollo-server` and the `GraphQLScalarType` from `graphql` at the top of `DateTimeType.js`. We'll also instantiate a new `GraphQLScalarType` called `DateTimeType`:

server/src/graphql/scalars/DateTimeType.js

```
import { ApolloError } from "apollo-server";
import { GraphQLScalarType } from "graphql";
import validator from "validator";

const DateTimeType = new GraphQLScalarType({
  name: "DateTime",
  description: "An ISO 8601-encoded UTC date string."
  // date string validation logic will go here...
});

export default DateTimeType;
```

The `GraphQLScalarType` expects us to pass in an object with a `name` and a `description` for the custom Scalar type. In addition to these properties, we have to create three methods that set the rules for the expected input and output values for this Scalar. For our use case:

- `parseValue` will ensure the value sent *from* the client is a valid date string
- `serialize` will ensure the value to be sent *to* the client is a valid date string
- `parseLiteral` will ensure the value in the GraphQL abstract syntax tree (AST) is a valid date string

To do this validation, our final `DateTimeType` will look like this:

server/src/graphql/scalars/DateTimeType.js

```
import { ApolloError } from "apollo-server";
import { GraphQLScalarType } from "graphql";
import validator from "validator";

const DateTimeType = new GraphQLScalarType({
  name: "DateTime",
  description: "An ISO 8601-encoded UTC date string.",
  parseValue: value => {
    if (validator.isISO8601(value)) {
      return value;
    }
    throw new ApolloError("DateTime must be a valid ISO 8601 date string");
  },
  serialize: value => {
    if (validator.isISO8601(value)) {
      return value;
    }
  }
});
```

```
    throw new ApolloError("DateTime must be a valid ISO 8601 date string");
},
parseLiteral: ast => {
  if (validator.isISO8601(ast.value)) {
    return ast.value;
  }
  throw new ApolloError("DateTime must be a valid ISO 8601 date string");
}
);

export default DateTimeType;
```

Next, we'll import the new `DateTimeType` object into `resolvers.js` to handle the `DateTime` type as it appears in our schema:

`server/src/graphql/resolvers.js`

```
import DateTimeType from "./scalars/DateTimeType.js";

const resolvers = {
  DateTime: DateTimeType,
  // ...
}

export default resolvers;
```

Try rerunning the `review` query. You will see a new `DateTime` type in your GraphQL IDE, and the query should work just as it did before.

## Add a Rating Scalar

Let's take what we just learned about creating custom Scalar types and apply it to something a bit more complex now. We currently have a `rating` field on the `Review`, `CreateReviewInput`, and `UpdateReviewInput` Input types, and the output type of these fields is an `Int`. The value of this field is only supposed to be an integer from one to five (representing the number of stars the user rated the book), but we're not doing anything to enforce that requirement. We could check the value of the `rating` argument in the `createReview` and `updateReview` methods in the `JsonServerApi` data source, but a better, more expressive approach would be to codify this requirement in the schema. To do that, we'll add a `Rating` Scalar type and update all of the `rating` fields in the schema to use it:

server/src/graphql/typeDefs.js

```
import { gql } from "apollo-server";

const typeDefs = gql`  
  # ...  
  
  """  
  An integer-based rating from 1 (low) to 5 (high).  
  """  
  scalar Rating  
  
  # ...  
  
  """  
  Provides data to create a review.  
  """  
  input CreateReviewInput {  
    # ...  
    "The user's integer-based rating of the book (from 1 to 5)." rating: Rating!  
    # ...  
  }  
  
  # ...  
  
  """  
  Provides data to update a review.  
  """  
  input UpdateReviewInput {  
    # ...  
    "The user's integer-based rating of the book (from 1 to 5)." rating: Rating!  
    # ...  
  }  
  
  # ...  
  
  """  
  A user-submitted assessment of a book that may include a numerical rating.  
  """
```

```
type Review {  
  # ...  
  "The user's integer-based rating of the book (from 1 to 5)."  
  rating: Rating!  
  # ...  
}  
  
# ...  
;  
  
export default typeDefs;
```

We'll need to create the `GraphQLScalarType` for the `Rating` type next, so we'll create a `RatingType.js` file in the same `scalars` directory that we created in the previous section. The set-up will be very similar to the `DateTime` scalar, but this time we'll create our own `isValidRating` function instead of using the `validator` import:

`server/src/graphql/scalars/RatingType.js`

```
import { ApolloError } from "apollo-server";  
import { GraphQLScalarType } from "graphql";  
import { Kind } from "graphql";  
  
function isValidRating(value) {  
  return Number.isInteger(value) && value >= 1 && value <= 5;  
}  
  
const RatingType = new GraphQLScalarType({  
  name: "Rating",  
  description: "An integer representing a user rating from 1 and 5,  
    inclusive.",  
  parseValue: value => {  
    if (isValidRating(value)) {  
      return value;  
    }  
    throw new ApolloError("Rating must be an integer from 1 and 5");  
  },  
  serialize: value => {  
    if (isValidRating(parseInt(value))) {  
      return value;  
    }  
    throw new ApolloError("Rating must be an integer from 1 and 5");  
};
```

```
},
parseLiteral: ast => {
  const intValue = parseInt(ast.value);
  if (ast.kind === Kind.INT && isValidRating(intValue)) {
    return intValue;
  }
  throw new ApolloError("Rating must be and integer from 1 and 5");
};

export default RatingType;
```

Lastly, we'll add the RatingType to the resolvers:

*server/src/graphql/resolvers.js*

```
import DateTimeType from "./scalars/DateTimeType.js";
import RatingType from "./scalars/RatingType.js";

const resolvers = {
  DateTime: DateTimeType,
  Rating: RatingType,
  // ...
}

export default resolvers;
```

Once again, run a mutation to create a new review for a book (or update an existing one) and try submitting an invalid integer. You should see the following error response:

*API Response*

```
{
  "errors": [
    {
      "message": "Variable \"$input\" got invalid value 6 at
                  \"input.rating\"; Expected type \"Rating\". Rating must be an
                  integer from 1 and 5",
      // ...
    }
  ],
}
```

```
    "data": null
}
```

## Directives in GraphQL

In addition to custom Scalar types, we can further enhance the expressiveness, utility, and predictability of a GraphQL schema using *directives*. In GraphQL, directives are denoted by an `@` character and are used to annotate parts of a schema or an operation document so that a client, validator, or executor can take special actions in response to the presence of that directive.<sup>2</sup> Out-of-the-box, a GraphQL server must support the `@skip` and `@include` directives outlined in the specification, as well as an `@deprecated` directive if it supports an SDL-first approach to implementing type definitions. We can also define and use custom directives.

GraphQL has two different kinds of directives—there are *type system* directives (also known as *schema* directives) and *executable* directives (also known as *query* directives). Type system directives are used to annotate different elements of a schema written in SDL such as types, fields, and Enum values, while executable directives are used to annotate parts of an operation document such as fragments, field selections, and even the entire query, mutation, or subscription.

The built-in `@skip` and `@include` directives are examples of executable directives that can be used to selectively exclude or include fields in a query response based on some boolean condition. In this example with `@skip`, the `betaField` will only be included in the response if the `$useExperimental` value is false:

```
query GetAuthors($useExperimental: Boolean) {
  authors {
    name
    betaField @skip(if: $useExperimental)
  }
}
```

Conversely, using `@include` opts into querying a field if the `$useExperimental` value is true:

```
query GetAuthors($useExperimental: Boolean) {
  authors {
    name
    betaField @include(if: $useExperimental)
  }
}
```

---

<sup>2</sup><https://spec.graphql.org/June2018/#sec-Type-System.Directives>

As we can see, directives can also include required arguments inside of parentheses, though some may only have optional arguments or none at all.

The `@deprecated` directive, on the other hand, is a type system directive and is used to flag a field on any named type or an Enum type value that has been slated for removal from the schema so that clients can update their operations accordingly. It also supports a `reason` argument to provide more information about the deprecation:

```
type Author {  
  fullName: String  
  name: String @deprecated(reason: "Use `fullName`.")  
}
```

Tools such as Explorer will also display additional information in their user interface to bring attention to deprecated fields so developers will know to avoid using them. Because GraphQL schemas are designed to be evolved in response to new product requirements, using the `@deprecated` directive is an essential component of a field rollover strategy so fields that are no longer needed or supported can be pruned from the schema.

### Custom Executable Directives with Apollo Server

At the time of writing, the only executable directives supported by Apollo Server are the built-in `@skip` and `@include` directives. That means that it's only possible to define custom type system directives with Apollo Server. However, [Apollo Gateway](#) does provide support for defining custom executable directives for federated GraphQL APIs.

## Add an @unique Custom Directive

Now that we've seen what the built-in directives can do, it's time to make one of our own. To round out this chapter, we will implement a custom `@unique` type system directive that can be used on a field in an Input Object to ensure unique values are submitted for that field when creating or updating that value. As a result, we'll be able to enforce that users submit unique emails and usernames when creating a new user by using a generalized directive instead of explicit runtime logic in the `JsonServerApi` data source.

To use a custom directive in our schema, we must first define it. When we define a directive, we use the `directive` keyword and include its name preceded by an `@` character, and then specify any allowed arguments in parentheses. Lastly, we must specify the locations in the schema (or operation document for executable directives) where the directive may be used:

*server/src/graphql/typeDefs.js*

```
import { gql } from "apollo-server";

const typeDefs = gql`  

  directive @unique(  

    "The resource path name from the REST endpoint."  

    path: String!  

    "";  

    "The database key name upon which to force uniqueness.  

    If not provided, then the GraphQL schema field name will be used.  

    "";  

    key: String  

  ) on INPUT_FIELD_DEFINITION  

  # ...  

  `;  

  export default typeDefs;
```

Above, the `@unique` directive takes a `path` to specify what kind of REST resource we need to check (for example, `authors` or `users`) and a `key` argument to specify what field in `db.json` to check against. The `key` argument will only be needed if the property name in `db.json` is different from the name of the field in the GraphQL schema.

Additionally, because we're creating a directive to be used on input fields in the schema, we specify that this directive is valid on the `INPUT_FIELD_DEFINITION` location. You can view a full list of supported locations for executable directives<sup>3</sup> and type system directives<sup>4</sup> in the GraphQL specification. Before moving on from `typeDefs.js`, we'll annotate the `email` and `username` fields in the `SignUpInput` type to use the new `@unique` directive:

*server/src/graphql/typeDefs.js*

```
import { gql } from "apollo-server";  

  const typeDefs = gql`  

  # ...  

  "";  

  Provides data to create a user.
```

<sup>3</sup><https://spec.graphql.org/June2018/#ExecutableDirectiveLocation>

<sup>4</sup><https://spec.graphql.org/June2018/#TypeSystemDirectiveLocation>

```
"""
input SignUpInput {
  "The email address of the user (must be unique)."
  email: String! @unique(path: "users")
  "The full name of the user."
  name: String!
  "The user's chosen username (must be unique)."
  username: String! @unique(path: "users")
}

# ...
;

export default typeDefs;
```

Next, we need to write some code that will verify that the user-submitted values are unique for the `email` and `username` fields. To begin, we'll create a new directory called `directives` in `server/src/graphql` and add a `UniqueDirective.js` file to it. In this file, we'll import the `SchemaDirectiveVisitor` class from Apollo Server (which is provided by its underlying `graphql-tools` dependency). To implement a custom schema directive, we must override one of the visitor methods defined in this class. For the `@unique` directive, we will override the `visitInputFieldDefinition` method. The basic set-up will look like this:

`server/src/graphql/directives/UniqueDirective.js`

```
import { SchemaDirectiveVisitor } from "apollo-server";

class UniqueDirective extends SchemaDirectiveVisitor {
  visitInputFieldDefinition(field, { objectType }) {
    // Field-validating code goes here...
  }
}

export default UniqueDirective;
```

The `SchemaDirectiveVisitor` class is very powerful and there are numerous examples in the Apollo Server documentation that show how to use it to create different kinds of type system directives, ranging from basic to advanced. The code required to support the `@unique` directive is on the advanced side, so you may wish to review some of those examples before proceeding. See Apollo Server's directive documentation here:

<https://www.apollographql.com/docs/apollo-server/schema/creating-directives/>

Adding support for the `@unique` directive for a field on an Input Object type poses an interesting implementation challenge for us. With type system directives, it's a common practice to access a field's `resolve` property to alter the returned value for the field as required by the applied directive. However, there's no `resolve` property available for an Input Object's fields because they don't resolve like other fields do (because they describe inputs to an operation, rather than outputs). But for our custom directive to work, we need access to the value of any Input Object field that was annotated with the `@unique` directive at the time of a `Mutation` field's resolution to check if this value already exists in another resource in the database, and then throw an error if needed.

To do this, we can tap into the `resolve` property for any field on the `Mutation` type that uses an Input Object type as an argument (like the `SignUpInput` does). Ultimately, we'll need to hook into the resolver for any mutations that use the `SignUpInput` with the `@unique` directive applied to any of its fields to intercept the submitted values and determine if they are indeed unique. To do that, we'll need to create two helper methods—one to get all of the mutations from the schema based on some filtering callback, and another to get an argument value for a field in a mutation:

`server/src/graphql/directives/UniqueDirective.js`

```
import { SchemaDirectiveVisitor } from "apollo-server";

class UniqueDirective extends SchemaDirectiveVisitor {
  getMutations(predicate = null) {
    if (!this._mutations) {
      this._mutations = Object.values(
        this.schema.getMutationType().getFields()
      );
    }

    if (!predicate) {
      return this._mutations || [];
    }

    return this._mutations.filter(predicate);
  }

  getMutationArgumentValue(fieldName, args) {
    const argTuples = Object.entries(args);

    for (let i = 0; i < argTuples.length; i++) {
      if (argTuples[i][0] === fieldName) {
        return argTuples[i][1];
      }
    }
  }
}
```

```
const [key, value] = argTuples[i];

if (value !== null && typeof value === "object") {
  return this.getMutationArgumentValue(fieldName, value);
} else if (key === fieldName) {
  return value;
}
}

return null;
}

visitInputFieldDefinition(field, { objectType }) {
  // Field-validating code goes here...
}
}

export default UniqueDirective;
```

This approach was inspired by an issue comment in the GraphQL Tools repository. You can read more about the original solution here:

<https://github.com/ardatan/graphql-tools/issues/858#issuecomment-500170481>

In the code above, the `predicate` argument of `getMutations` is a function that must return `true` or `false` to filter the list of mutations in the `_mutations` property. We will use it to filter a list of mutations that use the `SignUpInput` for an argument type. The `getMutationArgumentValue` method helps us check the argument value of the field that must be unique (and recursively walks through any nested objects).

Now for the fun part. We're ready to build out the `visitInputFieldDefinition` method:

`server/src/graphql/directives/UniqueDirective.js`

```
import { defaultFieldResolver } from "graphql";
import { SchemaDirectiveVisitor, UserInputError } from "apollo-server";
import fetch from "node-fetch";

const baseUrl = process.env.REST_API_BASE_URL;

class UniqueDirective extends SchemaDirectiveVisitor {
```

```
// ...

visitInputFieldDefinition(field, { objectType }) {
  const { path, key } = this.args;
  const fieldName = key ? key : field.name;

  const mutationsForInput = this.getMutations(({ args = [] }) => {
    return args.find(arg => arg?.type?.ofType === objectType);
  });

  mutationsForInput.forEach(mutation => {
    const { resolve = defaultFieldResolver } = mutation;

    mutation.resolve = async (...args) => {
      const uniqueValue = this.getMutationArgumentValue(fieldName,
        args[1]);

      if (uniqueValue) {
        const response = await fetch(
          `${baseUrl}/${path}?${fieldName}=${uniqueValue}`);
      }
      const results = await response.json();

      if (results.length) {
        throw new UserInputError(`Value for ${fieldName} is already in
          use`);
      }
    }

    return await resolve.apply(this, args);
  );
});
}

export default UniqueDirective;
```

The new code may be a bit overwhelming at first, so let's step through it. We first extract the argument values set for the `@unique` directive on the field in the schema and set the `fieldName` to the name of the Input Object type's field by default if the optional `key` argument isn't provided. Next, we use the `getMutations` method we previously created to get a list of the mutations that use the Input Object type where the `@unique` directive has been applied to its fields. For our purposes, the `objectType` value will equal `SignUpInput` here.

Once we know what mutations use an Input Object type with the `@unique` directive applied to any of its fields, we get the value of the argument that needs to be unique and check if that value already exists for the resource in question in `db.json` by querying the REST API. If it already exists, we throw an error. Otherwise, we resolve the field as usual.

With this code in place, we can tidy up our `JsonServerApi` data source by removing the `checkUniqueUserData` method definition by removing the line where we previously called it in the `signUp` method and removing the `async` keyword from `signUp` as well:

`server/src/graphql/dataSources/JsonServerApi.js`

```
// ...

class JsonServerApi extends RESTDataSource {
    // ...

    signUp({ email, name, username }) {
        return this.post("/users", {
            createdAt: new Date().toISOString(),
            email,
            name,
            username
        });
    }

    // ...
}

export default JsonServerApi;
```

As the final step, we need to make Apollo Server aware of the logic we just implemented in the `UniqueDirective` class. Instead of adding it to our resolvers, we will instead update the `ApolloServer` configuration in the top-level `index.js` file to include a `schemaDirectives` property. The value of this property is an object where the properties are the names of the custom directives in the schema and the values are the classes that extend `SchemaDirectiveVisitor`:

`server/src/index.js`

```
// ...
import UniqueDirective from "./graphql/directives/UniqueDirective.js";

const server = new ApolloServer({
    // ...
    dataSources: () => {
```

```
    return {
      jsonServerApi: new JsonServerApi()
    };
},
schemaDirectives: [
  unique: UniqueDirective
]
});

// ...
```

Try running a `signUp` mutation again for a new user with an email address or username that's taken used by an existing user. You will now see either a `Value for email is already in use` or `Value for username is already in use` error message in the response. If unique values are submitted for each of these fields, then the mutation should execute successfully.

### Directive or Field Argument?

As a best practice, don't create custom directives for things that should be regular field arguments. Custom directives are helpful when you need to modify the output or behavior of more than one field in a generalized way, such as enforcing unique values or handling access control. Field arguments are excellent for altering behavior on a per-field basis. If you find yourself creating a custom directive to use on a single field, that's a good indicator that you may want to consider an argument instead.

Also, keep in mind that directives can be used on any locations that they support in their definition, so if the intention isn't to fully generalize how they can alter field output, then field arguments may be a better choice (or be prepared to deal with unintended use cases!).

## Summary

In this chapter, we added various enhancements to our schema, including documentation for types and fields, two custom Scalar types, and a custom directive to enforce unique values for Input Object fields. At this point, Bibliotech's GraphQL API has really taken shape and it's almost ready for use in a real client application. The only missing piece now is to secure the API through authentication and authorization, which will tackle in the next chapter.

## Chapter 6

# Authentication and Authorization

In this chapter, we will:

- Save passwords and sign a JSON Web Token for users when they sign up
- Add a `login` mutation that sends a token to a user
- Add middleware to Express to verify tokens sent with incoming requests
- Add a `viewer` query to obtain information about the currently authenticated user
- Add field-level authorization to a GraphQL API

## Locking Down a GraphQL API

After some hard work over the last five chapters, we now have a robust GraphQL API up and running for Bibliotech. That said, there are two important features still missing. First, while we have a basic `signUp` mutation, we don't have any way to log existing users into the application. The second missing feature is that a logged-in state for a user wouldn't mean anything in relation to how the API currently functions. Put differently, there's no way to limit a user's actions (such as updating their library) based on who the user is.

To support these features, we need to add *authentication* (a way to identify that a user is who they say they are) and *authorization* (a way to ensure users can only do the things that they're allowed to do) to the API. A good place to start when deciding to manage these concerns for the API would be the GraphQL specification itself to see what it has to say about how we should handle them. So on the topics of authentication and authorization, the GraphQL specification has this to say:

*Cue the sound of crickets chirping...*

That's right—the specification doesn't outline any particular requirements about how authentication and authorization are applied to a GraphQL API. So does that mean the world is our auth oyster? Technically it does, but in the years since GraphQL's public release numerous patterns and best practices have emerged that can inform our approach.

If you've added authentication to another application before, you may have used a session cookie or a JSON Web Token (JWT) to identify authenticated users. Both of these approaches can be used with GraphQL. GraphQL APIs also integrate well with authentication services such as [Auth0](#) or [Magic](#) in most cases. To secure Bibliotech, we'll implement token-based authentication with JWTs from scratch throughout this chapter.

### A Quick Note on Using Third-Party Authentication Services

Authentication is a hard problem to solve properly and securely, which is why third-party services exist solely to handle this concern for an application. These services often also provide support for advanced features that you usually wouldn't want to attempt to build on your own such as single sign-on (SSO) and multi-factor authentication.

While we're going to roll our own basic authentication solution in this chapter for educational purposes, for real-world applications you will likely want to investigate whether a third-party service would better suit your needs.

If you'd like to learn more about integrating Auth0 as an authentication solution for a GraphQL API, then check out [Advanced GraphQL with Apollo & React](#).

## What's a JSON Web Token?

Before we can set up token-based authentication with JWTs, we should learn a bit more about what they are and how they work. If you're already familiar with JWTs, then feel free to jump ahead to the next section.

JWTs may seem fancy at first glance, but they're merely a specific kind of token that conforms to an open standard that describes how information may be transmitted as a compact JSON object. To verify a user's identity when they make requests to our GraphQL server for protected data, the user's browser will need to send a valid JWT with every request (often in an `Authorization` header), otherwise, the user will need to re-authenticate.

JWTs consist of three distinct parts:

1. **Header:** Contains information about the token type and the algorithm used to sign the token (for example, HS256).
2. **Payload:** Contains claims about a particular entity, and these statements may have predefined meanings in the JWT specification (known as *registered* claims) or they can be defined by the JWT user (known as *public* and *private* claims).
3. **Signature:** Helps to verify that no information was changed during the token's transmission by hashing together the token header and its payload, along with a secret.

To create the token, each of these parts are base64url-encoded (for compactness) and then concatenated together with a period. A typical JWT may look like this:

```
eyJhbGciOiJSUzI1NiIsInR5cCI6IkpXVCJ9eyJdWIiOiIxMjM0NTY3ODkwIiwiaWF0IjoxNjTE2MjM5MDIyLCJuYW1lIjoiQm9iIFJvc3MiLCJwYludGVyIjp0cnVlfQ.IAHZ341NCj8ggBRNjirYcLcV1nQUAaQt_P-FWJI-utW5nYhRj0EzFp_pyhz5JlyGMKveTJYxYF2_YuM7mSHBdbcUpoUj-Z8JdqdoCifuddIcOL3ZaLCQhkgI84gf_T_u77a9PLjuRhExkYFjd73CTsXQUJgjn8YnKC8263jVsYktN6N7iFcvoWgWyRwdX4xWmyIES3LJWjwPW8cWINJICc6m3z488URHlvhldShlOTJCWT0hrjVaG0yFhSzqtDl8dU5_E6InMd3qyuQNus5TBiMacLcuLcP8GzpEnp3yCeRUCuhpw8ltDK0GVsMMjVNBoFWVEps5iDAG7zdY18GiJ5zvnLg
```

It's important to note that even though the JWT above may look encrypted it's just base64url-encoded, so all of the information inside can just as easily be decoded again. Similarly, the signature portion of the JWT only helps us ensure that its data hasn't been changed while in transmission between the sender and receiver. The signature plays no role in encrypting the information contained within. For these reasons, it's important to never put any secret information inside of a JWT header or payload in clear text.

The header and payload sections of the above token would respectively decode to:

#### JWT Header

```
{  
  "alg": "HS256",  
  "typ": "JWT"  
}
```

#### JWT Payload

```
{  
  "sub": "1234567890",  
  "iat": 1615147456,  
  "artist": "Bob Ross",  
  "medium": "oil paint"  
}
```

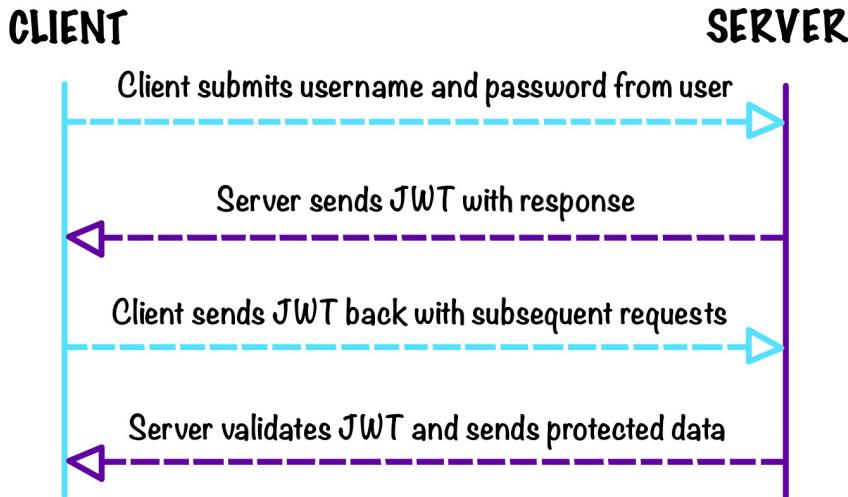
In the token's payload, the `sub` and `iat` claims represent *registered* claims, where `sub` (short for “subject”) is a unique identifier for the object described by the token. The `iat` claim is the time at which the token was issued. These claims are a part of the JWT specification. However, `artist` and `medium` are *custom* claims added to token, and specifically, they would be considered *private* claims and meant for use in a closed network only. By contrast, a public claim is also user-defined but would be registered with the IANA JSON Web Token Claims registry or have a specially formatted name (such as a URI) to avoid naming collisions. There are no public claims in the token above.

You can experiment with encoding and decoding JWTs at <https://jwt.io>.

Using a JWT like the one above, a typical user authentication flow would follow these steps:

1. A user would submit their username and password in a request from a client
2. The server would verify the submitted username and password against data saved in a database and then send a JWT back to the client to be used as an access token (until the token expires)
3. The client will send that token back in an `Authorization` header or a cookie with subsequent requests to the server
4. The server will verify the JWT and then send back the protected data to the user in its response if the JWT is valid

This process can be illustrated as follows:



After authenticating, the JWT has to be sent back with every request because HTTP is a stateless protocol, meaning that there's no built-in way for the response of one HTTP request to be altered based on the outcome of a previous request. In other words, if a user logs in and then navigates to another page or requests more data, they must send their provided JWT along with the second request because the server will have no memory of them making the first request to authenticate.

With this knowledge in hand, we're ready to do some light refactoring so we'll be ready to begin providing JWTs to authenticated Bibliotech users.

## Save a Hashed Password on User Sign-up

We're going to set up password-based authentication for users, so when they sign up we'll need to collect a password from them. Our first step will be to update the `SignUpInput` with a field for the password string:

`server/src/graphql/typeDefs.js`

```
import { gql } from "apollo-server";

const typeDefs = gql`  
# ...  
  
"""  
Provides data to create a user.  
"""  
input SignUpInput {  
# ...  
# ...  
  The user's chosen password.  
  
  It must be a minimum of 8 characters in length and contain 1 lowercase  
  letter, 1 uppercase letter, 1 number, and 1 special character.  
  """  
  password: String!  
  "The user's chosen username (must be unique)."  
  username: String! @unique(path: "users")  
}  
  
# ...  
`;  
  
export default typeDefs;
```

When a user signs up, it would be a good idea to return their token to them so they can subsequently send authenticated requests. At the moment, the `signUp` mutation only outputs a `User`, so let's wrap the `User` in an `AuthPayload` type and add `token` field to it, and then update the output type for the mutation:

`server/src/graphql/typeDefs.js`

```
import { gql } from "apollo-server";
```

```
const typeDefs = gql`  
  # ...  
  
  """  
  A currently authenticated user and their valid JWT.  
  """  
  type AuthPayload {  
    "The logged-in user."  
    viewer: User  
    "A JWT issued at the time of the user's most recent authentication."  
    token: String  
  }  
  
  # ...  
  
  type Mutation {  
    # ...  
    "Creates a new user."  
    signUp(input: SignUpInput!): AuthPayload!  
    # ...  
  }  
;  
  
export default typeDefs;
```

Next, we'll need to update the `signUp` method in the `JsonServerApi` data source to encrypt and store the user password (because we never want to store user passwords in clear text!). To do that, we can use the `crypto` module built into Node.js and we'll create some helper functions for hashing and verifying passwords in a `passwords.js` file in a new `utils` directory in `server/src`:

`server/src/utils/passwords.js`

```
import { promisify } from "util";  
import { randomBytes, scrypt, timingSafeEqual } from "crypto";  
  
const scryptAsync = promisify(scrypt);  
  
export async function hashPassword(password) {  
  const salt = randomBytes(16).toString("hex");  
  const derivedKey = await scryptAsync(password, salt, 64);  
  return salt + ":" + derivedKey.toString("hex");
```

```
}

export async function verifyPassword(password, hash) {
  const [salt, key] = hash.split(":");
  const keyBuffer = Buffer.from(key, "hex");
  const derivedKey = await scryptAsync(password, salt, 64);
  return timingSafeEqual(keyBuffer, derivedKey);
}
```

In the `hashPassword` function above, we use a promisified version of the `scrypt` method to create a salted hash of the user's password. The `verifyPassword` function extracts the salt and verifies that the submitted password's hash matches the saved password's hash. We also use `timingSafeEqual` to help protect against timing attacks when comparing the two hashes.

Now we'll import the two new password functions into `JsonServerApi.js`, and we'll also use the previously installed `validator.js` package to check if a user submits a strong password when they sign up:

`server/src/graphql/dataSources/JsonServerApi.js`

```
import { ForbiddenError, UserInputError } from "apollo-server";
import { RESTDataSource } from "apollo-datasource-rest";
import parseLinkHeader from "parse-link-header";
import validator from "validator";

import { hashPassword, verifyPassword } from "../../utils/passwords.js";

// ...
```

Next, we'll update the `signUp` method to confirm password strength and validate the password. We also mark the method as `async` so we can wait for the `hashPassword` promise to resolve. Note that the `validate.strongPassword` method applies some default password strength criteria, but we could pass in parameters to this function if we wanted to adjust these requirements:

`server/src/graphql/dataSources/JsonServerApi.js`

```
// ...

class JsonServerApi extends RESTDataSource {
  // ...

  async signUp({ email, name, password, username }) {
    if (!validator.strongPassword(password)) {
```

```
        throw new UserInputError(
            "Password must be a minimum of 8 characters in length and contain 1
             lowercase letter, 1 uppercase letter, 1 number, and 1 special
             character."
        );
    }

    const passwordHash = await hashPassword(password);

    return this.post("/users", {
        email,
        name,
        password: passwordHash,
        username
    });
}

// ...

export default JsonServerApi;
```

As a bonus challenge, you may wish to apply what you learned in the last chapter about custom Scalar types to implement a `Password` type that enforces strong password requirements instead of applying this logic in the data source method.

The new and improved `signUp` implementation is shaping up, but if we tried to run the mutation it would fail. At the moment, we're still only returning the user data, but the schema expects an object in the shape of the new `AuthPayload` type. To fix this, we'll need to generate a JWT to send along with the user data in the response. We'll use the `jsonwebtoken` package in the `server` directory to facilitate JWT creation:

```
npm i jsonwebtoken@8.5.1
```

Recall that a JWT contains three parts: a header, a payload, and a signature. The signature helps ensure that the JWT isn't tampered with before it's returned to the server in a subsequent request. To generate the signature using a symmetric signing algorithm ([HMAC with SHA-256](#), in our case), we have to provide a secret that must be known at the point that a JWT is created and elsewhere where the JWT is verified. The secret should be a hard-to-guess string and should be kept somewhere safe (and not committed to version control). For Bibliotech, we'll create a random string for the secret and add it as an environment variable in the `.env` file:

server/.env

```
JWT_SECRET=your_random_secret_here
NODE_ENV=development
REST_API_BASE_URL=http://localhost:5000
```

Now we can modify the `signUp` method a final time to create the signed JWT and return an object containing the token and the new user's data:

server/src/graphql/dataSources/JsonServerApi.js

```
// ...
import jwt from "jsonwebtoken";
import parseLinkHeader from "parse-link-header";
import validator from "validator";

class JsonServerApi extends RESTDataSource {
    // ...

    async signUp({ email, name, password, username }) {
        if (!validator.isStrongPassword(password)) {
            throw new UserInputError(
                "Password must be a minimum of 8 characters in length and contain 1 lowercase letter, 1 uppercase letter, 1 number, and 1 special character"
            );
        }

        const passwordHash = await hashPassword(password);
        const user = await this.post("/users", {
            email,
            name,
            password: passwordHash,
            username
        });
        const token = jwt.sign({ username }, process.env.JWT_SECRET, {
            algorithm: "HS256",
            subject: user.id.toString(),
            expiresIn: "1d"
        });

        return { token, viewer: user };
    }
}
```

```
// ...
}

export default JsonServerApi;
```

In the code above, the first argument passed into `jwt.sign` is an object containing any custom payload data. For our purposes, it will be sufficient to only add the user's username. The signing algorithm, subject, and expiration time are included in the standard JWT options passed in as the third argument.

Let's try running the `signUp` mutation again to confirm that it hashes and store the user's password in `db.json` and returns the JWT with the user data:

#### *GraphQL Mutation*

```
mutation SignUp($input: SignUpInput!) {
  signUp(input: $input) {
    token
    viewer {
      id
      email
      name
      username
    }
  }
}
```

#### *Mutation Variables*

```
{
  "input": {
    "email": "scout@email.com",
    "name": "Scout Finch",
    "username": "mockingbird60",
    "password": "superHARDpa55!"
  }
}
```

#### *API Response*

```
{
  "data": {
```

```
"signUp": {  
  "token": "eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJ1c2VybmFtZSI6Im1vYj  
  2tpbmdiaXJkNjAiLCJpYXQiOjE2MTUxNDg3OTEsImV4cCI6MTYxNTIzNTE5MSwi3V  
  iIjoiMyJ9.E2w4IXbTZg8sxz_1iFiZSV2wYsH0wGRFzce77G6Eg6o",  
  "viewer": {  
    "id": "3",  
    "email": "scout@email.com",  
    "name": "Scout Finch",  
    "username": "mockingbird60"  
  }  
}  
}  
}
```

After successfully running the update mutation, try copying the `token` value into the JWT debugger on <https://jwt.io> to confirm that it can be decoded correctly.

## Add a login Mutation

To complement the `signUp` mutation, we'll add a `login` mutation to allow existing users to log back into Bibliotech using their username and password. The first step will be adding a `login` field to the `Mutation` type in the schema. This field will use the same `AuthPayload` output type as the `signUp` mutation does:

`server/src/graphql/typeDefs.js`

```
import { gql } from "apollo-server";  
  
const typeDefs = gql`  
  # ...  
  
  type Mutation {  
    # ...  
    "Authenticates an existing user."  
    login(password: String!, username: String!): AuthPayload!  
    # ...  
  }  
`;  
  
export default typeDefs;
```

Next, we need a supporting `login` method in the `JsonServerApi` data source that will fetch a user resource based on the submitted username, verify that the hashed password stored in `db.json` matches the hash of the password string submitted with the mutation, sign a JWT for the user if the password is correct, and then return the user data and the token:

`server/src/graphql/dataSources/JsonServerApi.js`

```
import {
  AuthenticationError,
  ForbiddenError,
  UserInputError
} from "apollo-server";
// ...

class JsonServerApi extends RESTDataSource {
  // ...

  async login({ password, username }) {
    const user = await this.getUser(username);

    if (!user) {
      throw new AuthenticationError(
        "User with that username does not exist"
      );
    }

    const isValidPassword = await verifyPassword(password, user.password);

    if (!isValidPassword) {
      throw new AuthenticationError("Username or password is incorrect");
    }

    const token = jwt.sign({ username }, process.env.JWT_SECRET, {
      algorithm: "HS256",
      subject: user.id.toString(),
      expiresIn: "1d"
    });

    return { token, viewer: user };
  }
}

export default JsonServerApi;
```

Lastly, we'll add the new resolver for the `login` field:

`server/src/graphql/resolvers.js`

```
// ...

const resolvers = {
  // ...
  Mutation: {
    // ...
    login(root, args, { dataSources }, info) {
      return dataSources.jsonServerApi.login(args);
    },
    // ...
  }
};

export default resolvers;
```

Try running a `login` mutation now to authenticate an existing user:

*GraphQL Mutation*

```
mutation Login($password: String!, $username: String!) {
  login(password: $password, username: $username) {
    token
    viewer {
      id
      email
      name
      username
    }
  }
}
```

*Mutation Variables*

```
{
  "username": "mockingbird60",
  "password": "superHARDpa55!"
}
```

*API Response*

```
{  
  "data": {  
    "login": {  
      "token": "eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJ1c2VybmFtZSI6Im1vYj2tpbmdiaXJkNjAiLCJpYXQiOjE2MTUxNDg5NjIsImV4cCI6MTYxNTIzMiwic3ViIjoiMyJ9.M5MRuuPLOWMyfNxEyzfqMRKzyPA4RpizhL9d706KlrE",  
      "viewer": {  
        "id": "3",  
        "email": "scout@email.com",  
        "name": "Scout Finch",  
        "username": "mockingbird60"  
      }  
    }  
  }  
}
```

Once again, you can try decoding the token on <https://jwt.io> to confirm that the token in the response is structured as expected.

## Migrate to Apollo Server Express

We can successfully issue JWTs to authenticated users now, but we don't have any code in place to verify those tokens when they're sent back to the server on subsequent requests yet. Ideally, incoming tokens will be intercepted and validated in a middleware in our server before GraphQL execution begins (because we don't want to verify tokens independently in each field resolver function!). To accomplish this, we'll upgrade from the `apollo-server` package we've used so far to the `apollo-server-express` package instead. Apollo Server uses Express under the hood, but using `apollo-server-express` will allow us to tap into the Express middleware API directly.

Apollo Server also supports [integrations for many other Node.js web frameworks](#), including Koa, hapi, and Fastify.

First, we'll install the Express version of Apollo Server, the Express package itself, and the CORS middleware package:

```
npm i apollo-server-express@2.22.2 cors@2.8.5 express@4.17.1
```

Now we'll need to update the Apollo Server imports in several files:

server/src/graphql/typeDefs.js

```
import { gql } from "apollo-server-express";

const typeDefs = gql`  
  # ...  
`;  
  
export default typeDefs;
```

server/src/graphql/dataSources/JsonServerApi.js

```
import {  
  AuthenticationError,  
  ForbiddenError,  
  UserInputError  
} from "apollo-server-express";  
import { RESTDataSource } from "apollo-datasource-rest";  
import jwt from "jsonwebtoken";  
import parseLinkHeader from "parse-link-header";  
import validator from "validator";  
  
// ...
```

server/src/graphql/directives/UniqueDirective.js

```
import { defaultFieldResolver } from "graphql";  
import { SchemaDirectiveVisitor, UserInputError } from  
  "apollo-server-express";  
import fetch from "node-fetch";  
  
// ...
```

server/src/graphql/scalars/DateTimeType.js

```
import { ApolloError } from "apollo-server-express";  
import { GraphQLScalarType } from "graphql";  
import validator from "validator";  
  
// ...
```

*server/src/graphql/scalars/RatingType.js*

```
import { ApolloError } from "apollo-server-express";
import { GraphQLScalarType } from "graphql";
import { Kind } from "graphql";

// ...
```

Before we set up Express, we'll add a new environment variable for the port on which we expose the GraphQL endpoint:

*server/.env*

```
GRAPHQL_API_PORT=4000
JWT_SECRET=your_random_secret_here
NODE_ENV=development
REST_API_BASE_URL=http://localhost:5000
```

To configure Express, we'll need to make some changes to the main `index.js` file. First, we'll need to import `cors` and `express` and then create an Express application. If you're using Explorer as your GraphQL IDE, you'll need to allow the Apollo Studio URL as an origin. We will also preemptively enable CORS for `http://localhost:3000` because this is the port we'll run the React application on in the next chapter. Finally, instead of calling the `listen` method on the `server`, we now apply the Express instance and the CORS middleware to the server, then call a `listen` method on the Express app instead:

*server/src/index.js*

```
import { ApolloServer } from "apollo-server-express";
import cors from "cors";
import express from "express";

// ...

const port = process.env.GRAPHQL_API_PORT;
const app = express();

if (process.env.NODE_ENV === "development") {
  app.use(
    cors({
      origin: ["https://studio.apollographql.com", "http://localhost:3000"]
    })
);
```

```
}

const server = new ApolloServer({
  // ...
});

server.applyMiddleware({ app, cors: false });

app.listen({ port }, () =>
  console.log(`Server ready at
  http://localhost:${port}${server.graphqlPath}`)
);
```

Note that we set `cors: false` on the Apollo Server here to override its CORS defaults so we can use our custom CORS settings instead. If we restart the server now, we should see that everything works as it did before.

## Set Valid, Decoded JWTs on Apollo Server's Context

Now that we're up and running with Express, we can take advantage of a ready-made middleware package that will validate an incoming JWT, decode it, and add that value to the `req` object as a `user` property. To begin, we'll install `express-jwt` in the `server` directory:

```
npm i express-jwt@6.0.0
```

Next, we'll add the middleware to the main `index.js` file, using the same secret that was used to sign the JWT in the `signUp` and `login` mutations, choosing the same signing algorithm, and setting the `credentialsRequired` option to `false` so Express won't throw an error if a JWT hasn't been included (which would be the case for requests from the React application the don't require authentication or when Explorer polls for schema updates):

`server/src/index.js`

```
// ...
import expressJwt from "express-jwt";
// ...

app.use(
  expressJwt({
    secret: process.env.JWT_SECRET,
```

```
    algorithms: ["HS256"],
    credentialsRequired: false
}),
(err, req, res, next) => {
  if (err.code === "invalid_token") {
    return next();
  }
  return next(err);
};

const server = new ApolloServer({
  // ...
});

// ...
```

The second argument to `app.use` adds some custom error-handling logic when managing authorized access to the Express application. We must do this because by default `express-jwt` will throw an error if a token is invalid, even with `credentialsRequired` set to `false`. We don't want Express to throw an error at this point in the request execution if the user submits an invalid token (for instance, if the token has exceeded its expiration time), so we must do some extra error handling here.

Once the decoded JWT is available on the request object, we need to make Apollo Server aware of it so the token data can be used by resolvers to determine if a user is allowed to make a particular request to the API. With Apollo Server, it's a common practice to add decoded JWTs to its `context` object because this object is conveniently available in every resolver function as the third parameter. The `context` object is recreated with every request so we won't have to worry about access tokens going stale either.

To add data to this object, we'll set the `context` property on the Apollo Server that we instantiate. This property's value can either be an object or a function that returns an object. We'll use a function as a value here because this function's parameter will have access to the `req` object from Express and we can access the token that was decoded and validated by the Express middleware on the `user` property of that object:

`server/src/index.js`

```
// ...

const server = new ApolloServer({
  typeDefs,
```

```
resolvers,
dataSources: () => {
  return {
    jsonServerApi: new JsonServerApi()
  };
},
schemaDirectives: {
  unique: UniqueDirective
},
context: ({ req }) => {
  const user = req.user || null;
  return { user };
}
);
// ...
```

With this code in place, whenever a valid token is submitted in an `Authorization` header on a request to the Bibliotech API, every resolver function will be able to access data about that user.

## Add a viewer Query

It's time to put that user data to use now! We're going to add a final field to the `Query` type that will return data about an authenticated user. To do that, we'll first add a `viewer` field to the schema:

`server/src/graphql/typeDefs.js`

```
import { gql } from "apollo-server-express";

const typeDefs = gql`  
# ...  
  
type Query {  
  # ...  
  "Retrieves the currently authenticated user."  
  viewer: User  
}  
  
# ...  
`;
```

```
export default typeDefs;
```

We won't need to add a new method to the `JsonServerApi` data source to help resolve this field. Instead, we can use the existing `getUser` method in the `viewer` resolver, but we'll destructure the `user` out of the `context` parameter and get the `username` value from the decoded JWT instead of getting the user's `username` value from the `args` parameter:

`server/src/graphql/resolvers.js`

```
// ...

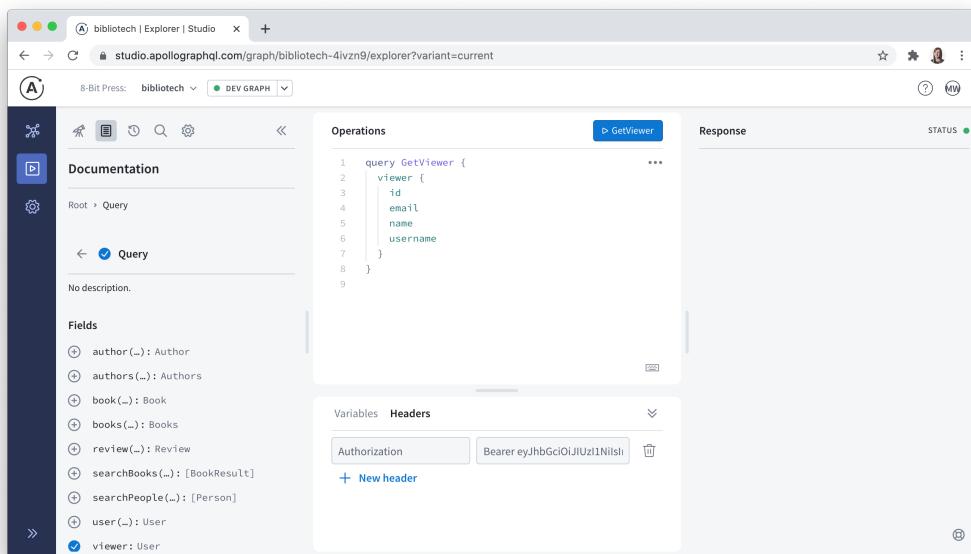
const resolvers = {
  // ...
  Query: {
    // ...
    user(root, { username }, { dataSources }, info) {
      return dataSources.jsonServerApi.getUser(username);
    },
    viewer(root, args, { dataSources, user }, info) {
      if (user?.username) {
        return dataSources.jsonServerApi.getUser(user.username);
      }
      return null;
    }
  },
  // ...
};

export default resolvers;
```

To test out this mutation, we'll need to add a JWT to the `Authorization` header of the request sent from Explorer (similar steps may be followed for other GraphQL IDEs as well). In JSON format, the `Authorization` header must be structured as follows with the word `Bearer` in front of the token and a space in between:

```
{
  "Authorization": "Bearer eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJ1c2VybmJFtZSI6Im1vY2tpbmdiaXJkNjAiLCJpYXQiOjE2MTUxNDg5NjIsImV4cCI6MTYxNTIzNTM2JMiwic3ViIjoiMyJ9.M5MRuuPLOWMyfNxEyzfqMRKzyPA4RpizhL9d706KlrE"
}
```

First, log in an existing user using the `login` mutation and then copy the `token` value from the response. Next, open the Headers panel and add the header name and value:



Now try running a `viewer` query to get information about the currently authenticated user based on that token:

#### GraphQL Query

```
query GetViewer {
  viewer {
    id
    email
    name
    username
  }
}
```

#### API Response

```
{
  "data": {
    "viewer": {
      "id": "3",
      "email": "user@example.com",
      "name": "John Doe",
      "username": "johndoe"
    }
  }
}
```

```
        "email": "scout@email.com",
        "name": "Scout Finch",
        "username": "mockingbird60"
    }
}
}
```

Ultimately, the `viewer` query will give us a more secure way to get information about a currently authenticated user, because the `username` value is retrieved from the signed JWT on the server, rather than relying on the client to send the correct `username` variable in a `user` query (which could potentially be tampered with by bad actors).

## Add GraphQL Shield for Authorization

We now have a way to identify users based on their JWTs, but we still don't have any mechanism to limit API access to authenticated users. Our final order of business in this chapter will be to add authorization to the Bibliotech GraphQL API so that users will only be able to perform the actions that the application's business rules allow.

We have a few options available for adding authorization to a GraphQL API. We could explicitly check the authenticated user's ID and any related permission data inside of each resolver and then throw an `AuthenticationError` as needed, but this wouldn't be very DRY. Alternatively, a popular option for adding authorization to GraphQL APIs involves adding [custom schema directives](#) to control access to various types and fields.

Yet another option is to abstract authorization into a separate layer and add it as resolver middleware, allowing us to check if a user is authorized to access a field when its resolver function is invoked. Just as we can apply middleware to an Express instance to intercept and alter its requests and responses, we can apply middleware directly to a GraphQL API to alter field resolution. This is the approach we'll choose and we'll implement it using a library called [GraphQL Shield](#).

We'll need to install two more packages to add authorization:

```
npm i graphql-middleware@6.0.4 graphql-shield@7.5.0
```

Next, we'll create a `permissions.js` file in the `server/src/graphql` directory to create a set of rules that check if a user is authorized to take specific actions before running field resolvers. First, we'll create a rule that checks if a user is authenticated and then apply that rule to the `user` and `searchPeople` queries because we only want registered Bibliotech users to be able to view other users:

server/src/graphql/permissions.js

```
import { rule, shield } from "graphql-shield";

const isAuthenticated = rule()(parent, args, { user }) => {
  return user !== null;
};

const permissions = shield(
{
  Query: {
    searchPeople: isAuthenticated,
    user: isAuthenticated
  },
  { debug: process.env.NODE_ENV === "development" }
};

export default permissions;
```

GraphQL Shield's `rule` function has all of the same parameters as a resolver function, so we can destructure the `user` object from the `context` parameter as we would in a resolver, and then check that the `user` is not `null`. Otherwise, we will return `false` to throw an authorization error for this rule.

To apply our new authorization rule, we must use the `applyMiddleware` function from GraphQL Middleware to add the `permissions` middleware to the schema. To use the `applyMiddleware` function on a schema, we must manually create our *executable schema* first (rather than allowing Apollo Server to do this for us), then apply our `permissions` middleware to it, and then pass the result into the `ApolloServer` constructor on as the `schema` option, rather than passing the `typeDefs` and `resolvers` into `ApolloServer` directly.

To do this, we need to update the imports at the top of the main `index.js` file:

server/src/index.js

```
import { ApolloServer, makeExecutableSchema } from "apollo-server-express";
import { applyMiddleware } from "graphql-middleware";
import cors from "cors";
import express from "express";
import expressJwt from "express-jwt";

import JsonServerApi from "./graphql/dataSources/JsonServerApi.js";
```

```
import permissions from "./graphql/permissions.js";
// ...
```

Next, we'll create a new `schema` variable and set it to the value of calling `makeExecutableSchema` on our `typeDefs`, `resolvers`, and `schemaDirectives`. We can then remove those same properties from the `ApolloServer` constructor, and instead set a `schema` property with a value that is the executable schema we just created with the `permissions` applied as middleware:

`server/src/index.js`

```
// ...

const schema = makeExecutableSchema({
  typeDefs,
  resolvers,
  schemaDirectives: {
    unique: UniqueDirective
  }
});
const schemaWithPermissions = applyMiddleware(schema, permissions);

const server = new ApolloServer({
  schema: schemaWithPermissions,
  dataSources: () => {
    return {
      jsonServerApi: new JsonServerApi()
    };
  },
  context: ({ req }) => {
    const user = req.user || null;
    return { user };
  }
});

// ...
```

After restarting the server, try running the following query without a valid JWT in the `Authorization` header:

*GraphQL Query*

```
query GetUser($username: String!) {
  user(username: $username) {
```

```
    id
    name
    username
  }
}
```

### Query Variables

```
{
  "username": "rabbithole84"
}
```

### API Response

```
{
  "errors": [
    {
      "message": "Not Authorised!",
      // ...
    }
  ],
  "data": {
    "user": null
  }
}
```

Try running the query again with a valid JWT in the header and you'll see that it returns the data as expected. We can also use the same `isAuthenticated` rule to prevent unauthenticated users from creating authors, books, and reviews:

`server/src/graphql/permissions.js`

```
import { rule, shield } from "graphql-shield";

// ...

const permissions = shield(
{
  Query: {
    searchPeople: isAuthenticated,
    user: isAuthenticated
  },
}
```

```
Mutation: {
  createAuthor: isAuthenticated,
  createBook: isAuthenticated,
  createReview: isAuthenticated
}
},
{ debug: process.env.NODE_ENV === "development" }
);

export default permissions;
```

Now let's add two additional rules to ensure that users can only edit resources that they own. The first will prevent users from adding or removing books from other users' libraries. To enforce this rule, we'll need to confirm that the `user.sub` value from the context matches the `userId` submitted in the mutation arguments:

*server/src/graphql/permissions.js*

```
import { rule, shield } from "graphql-shield";

// ...

const isUpdatingOwnLibrary = rule()(

  (root, { input: { userId } }, { user }, info) => {
    return user?.sub === userId;
  }
);

// ...
```

To apply both the `isAuthenticated` and `isUpdatingOwnLibrary` rules to a single field, we'll need to use GraphQL Shield's `and` function:

*server/src/graphql/permissions.js*

```
import { and, rule, shield } from "graphql-shield";

// ...

const permissions = shield(
  {
    // ...
  }
);
```

```
Mutation: {
  addBooksToLibrary: and(isAuthenticated, isUpdatingOwnLibrary),
  createAuthor: isAuthenticated,
  createBook: isAuthenticated,
  createReview: isAuthenticated,
  removeBooksFromLibrary: and(isAuthenticated, isUpdatingOwnLibrary)
}
},
{ debug: process.env.NODE_ENV === "development" }
);

export default permissions;
```

The second and final rule that we'll add will confirm that a user is updating or deleting their review. We can implement an `isEditingOwnReview` review as follows, using the `id` of the review to determine who owns it, and comparing it to the `user.sub` value:

`server/src/graphql/permissions.js`

```
import { and, rule, shield } from "graphql-shield";

// ...

const isEditingOwnReview = rule()(

  async (root, args, { dataSources, user }, info) => {
    const id = args.input ? args.input.id : args.id;
    const review = await dataSources.jsonServerApi.getReviewById(id);
    return user.sub === review.userId.toString();
  }
);

const permissions = shield(
{
  // ...
  Mutation: {
    // ...
    deleteReview: and(isAuthenticated, isEditingOwnReview),
    removeBooksFromLibrary: and(isAuthenticated, isUpdatingOwnLibrary),
    updateReview: and(isAuthenticated, isEditingOwnReview)
  },
  { debug: process.env.NODE_ENV === "development" }
}
```

```
);  
  
export default permissions;
```

Login as an existing user and try adding or removing books from another user's library and also try updating or deleting another user's reviews before moving on. You should see the `Not Authorised!` error thrown for each mutation, but still run the mutation successfully when editing the resources that the user owns.

## Summary

Throughout this chapter, we transformed the Bibliotech GraphQL API into a more production-ready state by adding user passwords on sign-up and creating JWTs for authentication purposes. We used Express middleware to validate incoming JWTs, and also added a new `login` mutation and `viewer` query to provide data about an authenticated user. Lastly, we added authorization to several fields using GraphQL Shield and applied those permissions to the API as middleware.

In the next chapter, we'll finally turn our attention to the client side and use Apollo Client to build out the new React application for Bibliotech.

## Chapter 7

# React App Set-up

In this chapter, we will:

- Bootstrap a React application and use Tailwind CSS to add a basic layout and styles
- Add React Router as a routing solution for the single-page React application
- Add routes for the index, home, and login pages
- Add a generalized layout to the index and home pages
- Add a form for existing users to log in and new users to sign up on the login page

## Create a React App

Bibliotech's GraphQL API is ready to go so it's finally time to start working on the client application. Our high-level plan to build out the front-end includes the following tools:

- [Create React App](#) to generate the boilerplate React application
- [Tailwind CSS](#) as a style library to rapidly build out a modern user interface
- [Apollo Client](#) to make queries to our GraphQL API

This chapter will focus on laying the groundwork for the Bibliotech client application, so we'll add the basic layout and create some initial routes for now, then shift our focus to Apollo Client in Chapter 8. To begin, we need a starter React application to work with, so we'll use the Create React App CLI tool to scaffold one. From the root directory of your project code (in other words, one level up from `server`), create a new React application using `create-react-app`:

```
npx create-react-app@4.0.3 client --use-npm
```

We'll continue using npm on the client side, so the `--use-npm` flag ensures Create React App uses npm to install dependencies even if Yarn has been installed locally. Next, we'll jump into the new `client` directory to see what was installed. The directory structure will look like this:

```
client
├── node_modules/
│   └── ...
├── public /
│   └── favicon.ico
│   └── index.html
│   └── logo192.png
│   └── logo512.png
│   └── manifest.json
│   └── robots.txt
└── src
    ├── App.css
    ├── App.js
    ├── App.test.js
    ├── index.css
    ├── index.js
    ├── logo.svg
    ├── reportWebVitals.js
    └── setupTests.js
├── .gitignore
├── package-lock.json
├── package.json
└── README.md
```

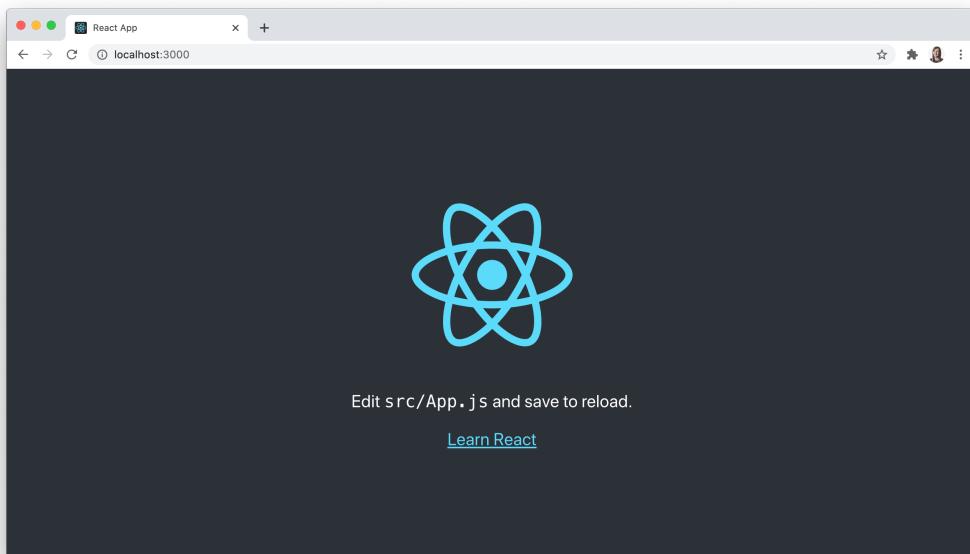
You can remove the `.gitignore` file and default `README.md` that were added here if you like. Next, if we take a look in `package.json`, we'll see that the `create-react-app` installation process installed three dependencies in our app (`react`, `react-dom`, and `react-scripts`) and we also have four different scripts we can run:

```
{
  // ...
  "scripts": {
    "start": "react-scripts start",
    "build": "react-scripts build",
    "test": "react-scripts test",
    "eject": "react-scripts eject"
  },
  // ...
}
```

Let's run the `start` script to start up the client application locally:

```
npm start
```

The React app will now be accessible at <http://localhost:3000>:



Now we'll strip down the starter files and code to the bare essentials. First, we'll delete the `App.js`, `App.css`, `App.test.js`, `logo.svg`, `reportWebVitals.js`, and `setupTests.js` files from `client/src`. Then we'll replace the starter code in `index.js` with the following:

`client/src/index.js`

```
import ReactDOM from "react-dom";

import "./index.css";

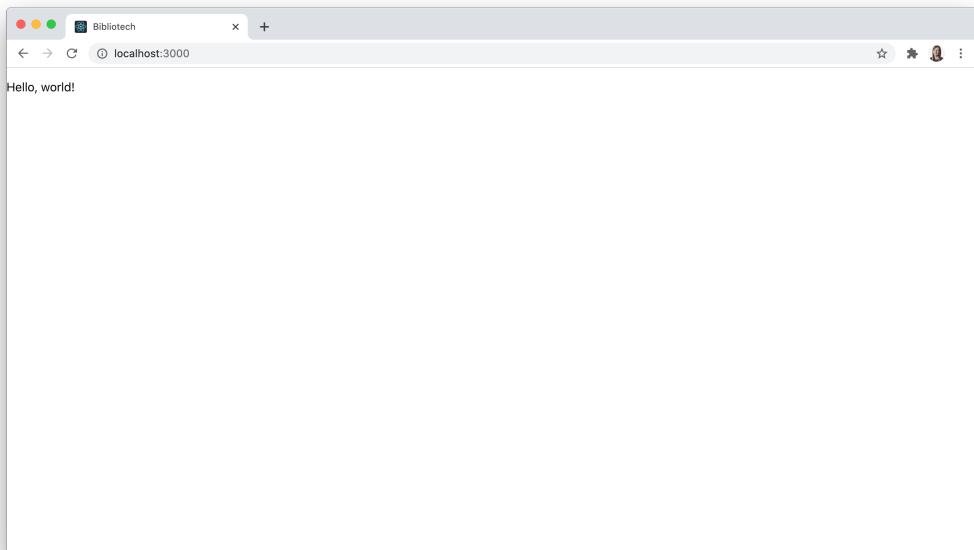
function App() {
  return (
    <div>
      <p>Hello, world!</p>
    </div>
  );
}

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

The revamped file and folder structure in `client` will look like this:

```
client
├── node_modules/
│   └── ...
├── public /
│   ├── favicon.ico
│   ├── index.html
│   ├── logo192.png
│   ├── logo512.png
│   └── manifest.json
│       └── robots.txt
└── src
    ├── index.css
    └── index.js
├── package-lock.json
└── package.json
```

We can also update the `title` element text in the `client/public/index.html` file to be something more relevant, such as “Bibliotech.” The running application will now look like this:



## Add Tailwind CSS

Using Tailwind to style our application's components will take some initial configuration. First, we need to install Tailwind and some PostCSS-related packages as development dependencies:

```
npm i -D tailwindcss@npm:@tailwindcss/postcss7-compat@2.0.3  
@tailwindcss/postcss7-compat@2.0.3 postcss@7.0.34 autoprefixer@9.8.6
```

Next, because Create React App doesn't have built-in support for overriding the PostCSS configuration, we'll have to install CRACO as a regular dependency:

```
npm i @craco/craco@6.1.1
```

The `craco` CLI must now be used in lieu of `react-scripts` for the `start`, `build`, and `test` scripts in the client's `package.json` file:

*client/package.json*

```
{  
  // ...  
  "scripts": {  
    "start": "craco start",  
    "build": "craco build",  
    "test": "craco test",  
    "eject": "react-scripts eject"  
  },  
  // ...  
}
```

To add the Tailwind and Autoprefixer PostCSS plugins, add a `craco.config.js` file to the `client` directory with the following code:

*client/craco.config.js*

```
module.exports = {  
  style: {  
    postcss: {  
      plugins: [require("tailwindcss"), require("autoprefixer")]  
    }  
  }  
};
```

From the `client` directory we'll also generate a `tailwind.config.js` file with the minimum amount of configuration using the `tailwindcss` CLI:

```
npx tailwindcss init
```

We should now have `tailwind.config.js` containing the following code:

*client/tailwind.config.js*

```
module.exports = {
  purge: [],
  darkMode: false, // or 'media' or 'class'
  theme: {
    extend: {},
  },
  variants: {
    extend: {},
  },
  plugins: [],
}
```

Let's update it to use the same system fonts as before:

*client/tailwind.config.js*

```
module.exports = {
  purge: [],
  darkMode: false, // or 'media' or 'class'
  theme: {
    extend: {},
    fontFamily: {
      sans: [
        "-apple-system, BlinkMacSystemFont, 'Segoe UI', 'Roboto', 'Oxygen',
        'Ubuntu', 'Cantarell', 'Fira Sans', 'Droid Sans', 'Helvetica
        Neue', sans-serif",
        mono: "source-code-pro, Menlo, Monaco, Consolas, 'Courier New',
        monospace"
      ],
    },
    variants: {
      extend: {},
    },
  },
}
```

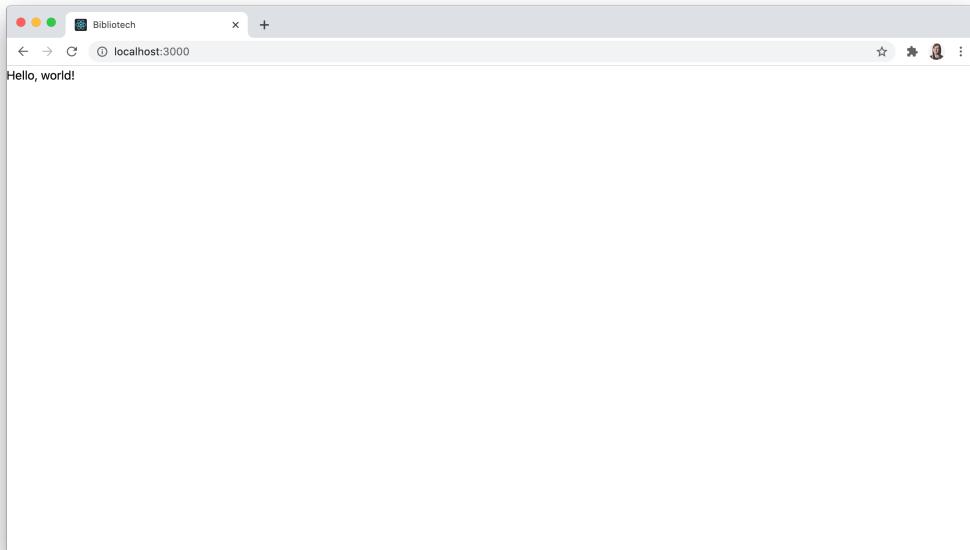
```
  plugins: [],  
};
```

Lastly, we'll replace the code in the existing `index.css` file with the following three lines to import the Tailwind styles:

`client/src/index.css`

```
@tailwind base;  
@tailwind components;  
@tailwind utilities;
```

After the React application restarts, it should look similar to what it did before, but with the top page margin reset now:



## Install React Router

Before we put Tailwind to work laying out our application, let's install React Router and configure a few basic routes. First, we'll install React Router in the `client` directory:

```
npm i react-router-dom@5.2.0
```

Next, we'll add a `pages` directory to `client/src`. Inside of `pages`, we'll create an `Index` directory to house the index page component and then add an `index.js` file to it with this code inside:

`client/src/pages/Index/index.js`

```
function Index() {
  return <p>This is the index page.</p>;
}

export default Index;
```

The index page will be what unauthenticated users see when they arrive at the application. Authenticated users will need to see a different page after they log in, so we'll also set up a `Home` directory in `pages` with an `index.js` file with this code in it:

`client/src/pages/Home/index.js`

```
function Home() {
  return <p>This is a user's homepage.</p>;
}

export default Home;
```

Lastly, users will need somewhere to go to log into the application, so we'll create `Login` page component too:

`client/src/pages/Login/index.js`

```
function Login() {
  return <p>This is the page where users authenticate.</p>;
}

export default Login;
```

Now we'll add a `router` directory to `client/src` with another `index.js` file and use it to define the initial routes for the application:

`client/src/router/index.js`

```
import { Route, Switch } from "react-router";

import Index from "../pages/Index";
import Home from "../pages/Home";
```

```
import Login from "../pages/Login";

export function Routes() {
  return (
    <Switch>
      <Route exact path="/" component={Index} />
      <Route exact path="/home" component={Home} />
      <Route exact path="/login" component={Login} />
    </Switch>
  );
}
```

We wrap our list of routes in a `Switch` component because it allows us to render a route exclusively. Without it, the `Route` component will render inclusively, meaning that more than one page could be rendered at a time in the user interface if its path matches. We also use the `exact` prop to ensure that the component for this route will only render if the path matches the `location.pathname` exactly (for example, all paths would match the top-level `/`, and we don't want `Index` to render everywhere).

Now we can update our `App` component in the top-level `index.js` file. We'll import the `BrowserRouter` from React Router and the `Routes` component first and then render the `Router` and `Routes` instead of the "Hello, world!" message:

`client/src/index.js`

```
import { BrowserRouter as Router } from "react-router-dom";
import ReactDOM from "react-dom";

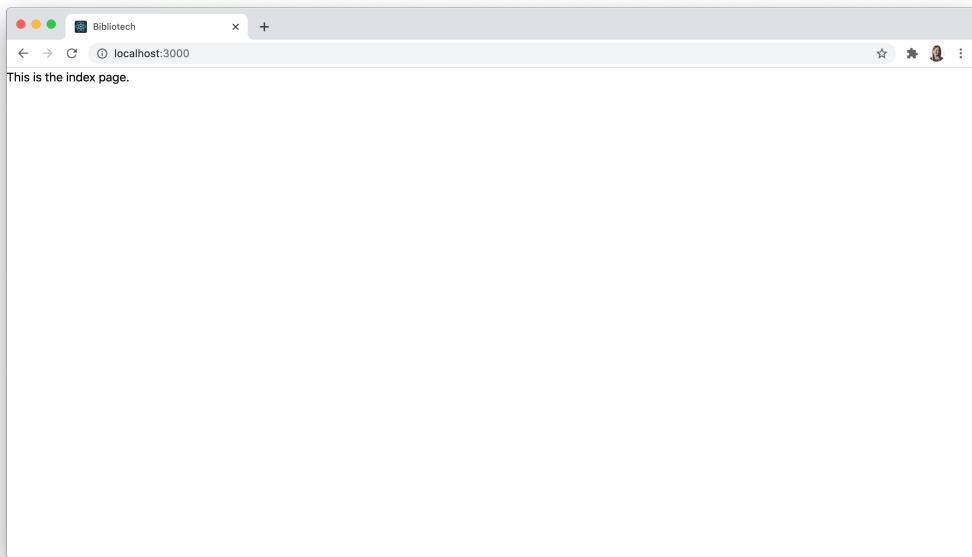
import { Routes } from "./router";

import "./index.css";

function App() {
  return (
    <Router>
      <Routes />
    </Router>
  );
}

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

We can see that our index page is rendered in the browser now:



Before moving on, navigate to `/home` and `/login` to ensure they render correctly.

## Style the Main Layout

Now back to styling. Tailwind provides a very powerful and robust collection of CSS classes that we'll use to layout and style our React components. While it may seem a bit intimidating at first glance, after a few minutes of working with Tailwind's classes you'll likely start to find this CSS framework relatively intuitive.

For example, to style a basic red button with white text and rounded corners, we would add the following classes to a `button` element:

```
<button  
  class="bg-red-500 hover:bg-red-700 font-semibold rounded px-4 py-2  
  text-white"  
>  
  Log In  
</button>
```

Try pasting this code into the [Tailwind playground](#) to see it in action. You can also try adding responsive styles by setting a base class for a particular element and then prefix additional classes with one of Tailwind's predefined breakpoint widths as follows: `text-sm md:text-lg`.

Tailwind has excellent documentation and it would be a good idea to take a quick look through it before proceeding with the rest of this chapter. If you’re ready, let’s jump in and use some Tailwind classes to style a reusable Button component that we’ll use for the “Log In” button in the application’s navigation bar. Add a `components` directory to `client/src` and add a `Button` directory inside of that with an `index.js` file containing the following code:

`client/src/components/Button/index.js`

```
function Button({ className, disabled, onClick, text, type }) {
  let buttonClasses =
    "bg-red-500 hover:bg-red-700 font-semibold rounded px-4 py-2 shadow
      hover:shadow-md focus:outline-none focus:shadow-outline text-white
      text-sm sm:text-base";

  if (className) {
    buttonClasses = `${buttonClasses} ${className}`;
  }

  if (disabled) {
    buttonClasses = `${buttonClasses} cursor-not-allowed`;
  }

  return (
    <button
      className={buttonClasses}
      disabled={disabled}
      onClick={onClick}
      type={type}>
      {text}
    </button>
  );
}

Button.defaultProps = {
  disabled: false,
  onClick: () => {},
  type: "button"
};

export default Button;
```

Button is the first component in this chapter to use props, and at a minimum, it would be a good idea to do some typechecking on any props we set up using React's PropTypes library. Alternatively, you may wish to use Flow or TypeScript too.

For brevity's sake, typechecking has been omitted from the client code in this book. You can read more about typechecking in the React documentation:

<https://reactjs.org/docs/typechecking-with-proptypes.html>

Next, we'll create a NavBar component to display the name of the application on the lefthand side of the header and a "Log In" button on the right:

*client/src/components/NavBar/index.js*

```
import { Link, useHistory } from "react-router-dom";

import Button from "../Button";

function NavBar() {
  const history = useHistory();

  return (
    <header className="bg-white border-b border-gray-200 border-solid">
      <div className="flex flex-wrap items-center justify-between mx-auto
        max-w-screen-lg px-8 py-8 w-full">
        <Link
          to="/"
          className="text-black hover:text-gray-800 hover:no-underline
          mr-4"
        >
          <h1>Bibliotech</h1>
        </Link>
        <div className="flex items-center sm:justify-end mt-2 sm:mt-0">
          <Button
            onClick={event => {
              event.preventDefault();
              history.push("/login");
            }}
            text="Log In"
          />
        </div>
      </div>
    </header>
```

```
    </header>
  );
}

export default NavBar;
```

Note that Tailwind completely resets the styles on all heading elements. In the code above, we could apply some styles directly to the `h1`, but we can also use Tailwind classes to style certain elements generically. To add some base styles to all `h1` elements in the application, add the following code in the `index.css` file directly after the `@tailwind base` import:

`client/src/index.css`

```
@tailwind base;

h1 {
  @apply font-bold leading-none text-5xl;
}

@tailwind components;
@tailwind utilities;
```

Now let's build up a `MainLayout` component using the `NavBar` for the application header, rendering the component's `children` in the main content area, and a `footer` element at the bottom of the page:

`client/src/components/MainLayout/index.js`

```
import NavBar from "../NavBar";

function MainLayout({ children }) {
  return (
    <div className="bg-gray-50">
      <div className="flex flex-col justify-between min-h-screen">
        <NavBar />
        <div className="flex-auto flex flex-col max-w-screen-lg mx-auto
          px-8 py-6 w-full">
          {children}
        </div>
        <footer className="bg-white border-t border-gray-200 border-solid
          py-4">
          <div className="">
```

```
<p className="text-center text-gray-700 text-sm sm:text-base">
    "A word after a word after a word is power."{ " "}
    <span className="whitespace-no-wrap">- Margaret Atwood</span>
</p>
</div>
</footer>
</div>
</div>
);

export default MainLayout;
```

With this code in place, we can import this component into the `Index` component file and wrap the content in the `MainLayout`:

`client/src/pages/Index/index.js`

```
import MainLayout from "../../components/MainLayout";

function Index() {
  return (
    <MainLayout>
      <p>This is the index page.</p>
    </MainLayout>
  );
}

export default Index;
```

And we can do the same for the `Home` component too:

`client/src/pages/Home/index.js`

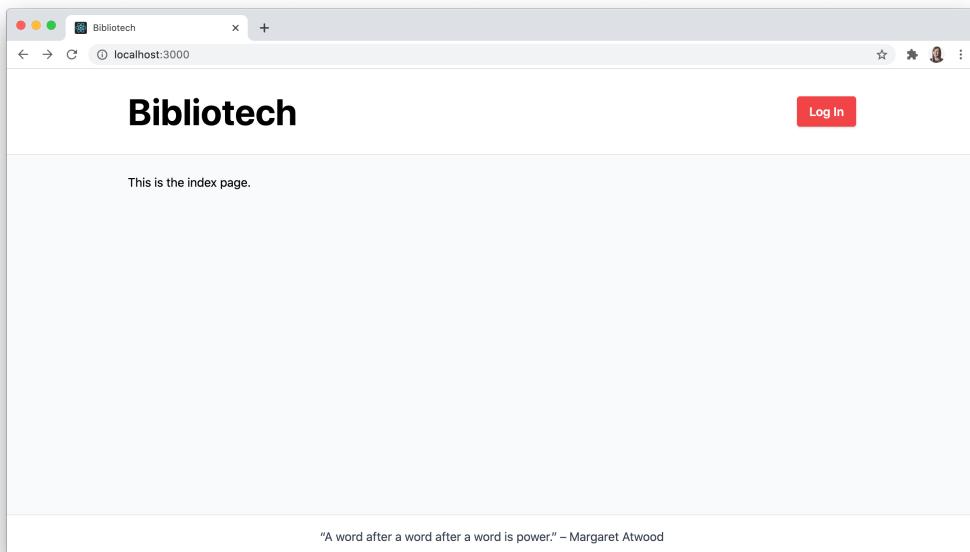
```
import MainLayout from "../../components/MainLayout";

function Home() {
  return (
    <MainLayout>
      <p>This is a user's homepage.</p>
    </MainLayout>
  );
}
```

```
}
```

```
export default Home;
```

The application should now render in the browser like this now:



## Style the Login Page Layout

We'll use the `MainLayout` component for most of our pages, but the `/login` route will have a unique layout applied to it. The layout will consist of a single form horizontally and vertically centered on a light grey background. As with the `Button` component in the previous section, it would be wise to create a reusable `TextInput` component to use throughout the application. We'll begin by creating a `TextInput` directory in `components` and adding the following code to its `index.js` file:

`client/src/components/TextInput/index.js`

```
function TextInput({  
  className,  
  error,  
  hiddenLabel,
```

```
    id,
    inputWidthClass,
    label,
    name,
    onChange,
    placeholder,
    type,
    value,
    ...rest
}) {
  return (
    <div className={className}>
      <label
        htmlFor={id}
        className={`block font-semibold ${hiddenLabel && "sr-only"}`}
      >
        {label}
      </label>
      <input
        className={`appearance-none bg-transparent focus:bg-gray-100
          border-b-2 border-red-500 focus:border-red-700 focus:outline-none
          mr-2 px-3 py-2 text-gray-800 text-sm sm:text-base
          ${inputWidthClass}`}
        id={id}
        onChange={onChange}
        type={type}
        placeholder={placeholder}
        value={value}
        {...rest}
      />
      {error && <p className="text-red-500 text-sm">{error}</p>}
    </div>
  );
}

TextInput.defaultProps = {
  inputWidthClass: "w-auto",
  type: "text"
};

export default TextInput;
```

Next, we'll import that new component and the `Button` in `Login` page component's file, remove the placeholder paragraph, and update it with the following content:

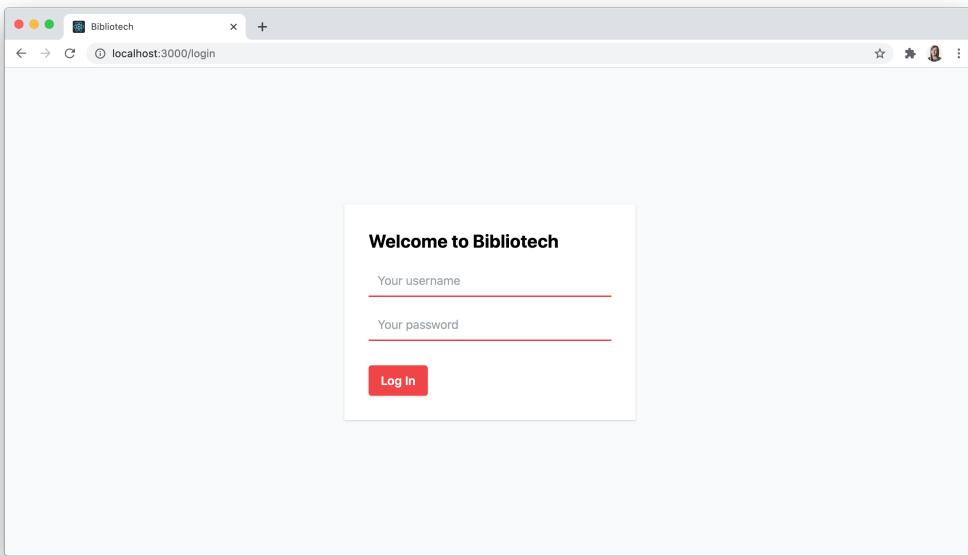
`client/src/pages/Login/index.js`

```
import Button from "../../components/Button";
import TextInput from "../../components/TextInput";

function Login() {
  return (
    <div className="bg-gray-50 flex items-center justify-center min-h-screen">
      <div className="bg-white shadow p-8 max-w-sm w-10/12">
        <h1 className="mb-4 text-2xl">Welcome to Bibliotech</h1>
        <form>
          <TextInput
            className="mb-4"
            hiddenLabel
            id="username"
            inputWidthClass="w-full"
            label="Username"
            placeholder="Your username"
            type="text"
          />
          <TextInput
            className="mb-8"
            hiddenLabel
            id="password"
            inputWidthClass="w-full"
            label="Password"
            placeholder="Your password"
            type="password"
          />
          <Button text="Log In" type="submit" />
        </form>
      </div>
    </div>
  );
}

export default Login;
```

The updated `/login` route should now render as follows:



Now we need to consider how users will initially sign up, and ideally, there will be some way to toggle the view of this form to work for new sign-ups as well. To do that, we'll need to manage some state. And to manage state, we'll need to use [React Hooks](#). If you're new to hooks, they offer a cleaner API to manage state in React components than what traditional lifecycle methods have afforded in a React Component subclass. To use hooks, we must keep in mind their two basic rules:

1. Only call hooks **at the top level** of a React component function (they can't be called from inside loops, conditionals, or other nested functions)
2. Only call hooks **from React components or within other custom hooks that you create** (so we can't call them from within regular JavaScript functions)

Using the `useState` hook will allow us to manage state conveniently within a component but without using a class-based component. When we call the `useState` hook, we can pass in an initial value for the state (or nothing at all) and it returns the current state value plus a function for updating that state in a tuple-like array. For this form, we'll need to add a boolean value to the component's state that tracks whether the user has identified as already being a member of Bibliotech. If they are, then we'll render fields to log in, otherwise, we'll render additional fields to facilitate the user sign-up process.

Let's import the `useState` hook from React and call it in the `Login` page component:

client/src/pages/Login/index.js

```
import { useState } from "react";

// ...

function Login() {
  const [isMember, setIsMember] = useState(true);

  // ...
}

export default Login;
```

Again, `isMember` is the current state value and `setIsMember` is a function that we can call to update that state to a new value. Let's use both now to conditionally render some of the existing text in the form so that it's relevant to new users. We'll toggle the state value back and forth when the user clicks on the new button:

client/src/pages/Login/index.js

```
// ...

function Login() {
  const [isMember, setIsMember] = useState(true);

  return (
    <div className="bg-gray-50 flex items-center justify-center min-h-screen">
      <div className="bg-white shadow p-8 max-w-sm w-10/12">
        <h1 className="mb-4 text-2xl">Welcome to Bibliotech</h1>
        <form>
          {/* ... */}
          <div className="flex items-center">
            <Button
              className="mr-2"
              text={isMember ? "Log In" : "Sign Up"}
              type="submit"
            />
            <p className="text-gray-400">
              {isMember ? "New here?" : "Already joined?"}
            </p>
            <button>
```

```
        className="text-red-500 hover:text-red-700 ml-1
          focus:outline-none hover:underline"
        onClick={event => {
          event.preventDefault();
          setIsMember(state => !state);
        }}
      >
    {isMember ? "Sign up." : "Log in."}
  </button>
</div>
</form>
</div>
</div>
);
}

export default Login;
```

Lastly, we'll conditionally render two additional fields in the form to capture a user's full name and email address when they sign up as well:

*client/src/pages/Login/index.js*

```
// ...

function Login() {
  const [isMember, setIsMember] = useState(true);

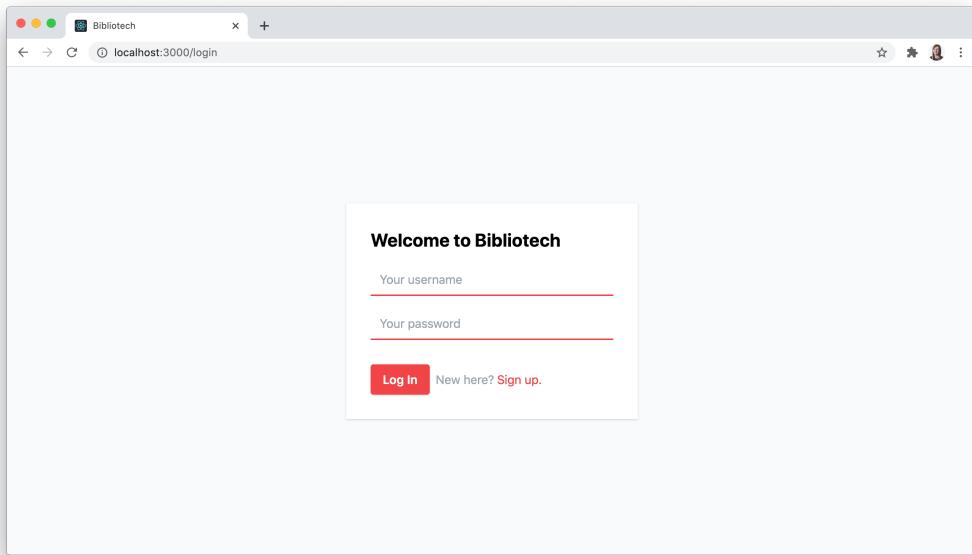
  return (
    <div className="bg-gray-50 flex items-center justify-center
      min-h-screen">
      <div className="bg-white shadow p-8 max-w-sm w-10/12">
        <h1 className="mb-4 text-2xl">Welcome to Bibliotech</h1>
        <form>
          <TextInput
            className="mb-4"
            hiddenLabel
            id="username"
            inputWidthClass="w-full"
            label="Username"
            placeholder="Your username"
            type="text"
          >
        </form>
      </div>
    </div>
  );
}

export default Login;
```

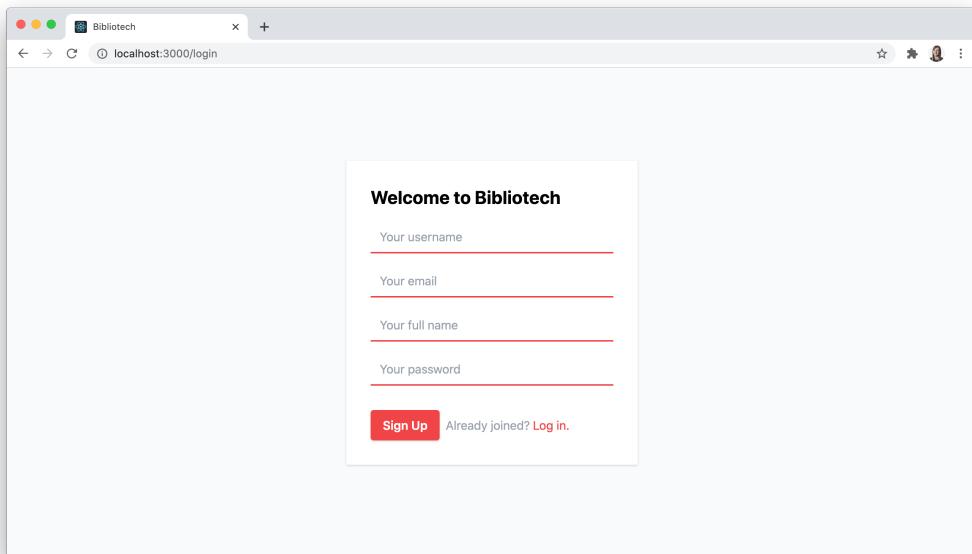
```
    />
  {!isMember && (
    <>
      <TextInput
        className="mb-4"
        hiddenLabel
        id="email"
        inputWidthClass="w-full"
        label="Email"
        placeholder="Your email"
        type="email"
      />
      <TextInput
        className="mb-4"
        hiddenLabel
        id="name"
        inputWidthClass="w-full"
        label="Full Name"
        placeholder="Your full name"
        type="text"
      />
    </>
  )}
  {/* ... */}
</form>
</div>
</div>
);
}

export default Login;
```

The form will now look like this when initially loaded:



And when the “Sign up” button is clicked, the form will re-render as follows:



## Summary

At the end of this chapter, we have a solid foundation in place for Bibliotech's new React application. We added the Tailwind CSS framework to style it and React Router to handle routing. We also applied a common layout to the index and home pages and roughed out a form that will allow existing users to log in or new users to sign up. In the next chapter, we'll finally add our GraphQL API into the mix by installing Apollo Client and sending queries to the API to populate the front-end application with real data.

## Chapter 8

# Apollo Client with User Authentication

In this chapter, we will:

- Install and configure Apollo Client in a React application
- Fetch a list of books to display on the index page
- Send a JWT to a client in an HttpOnly cookie
- Use React's context API to manage authentication state
- Add a logout button to the navigation bar
- Render private and public routes with React Router
- Fetch an authenticated user's library books and display them on the homepage

## Install Apollo Client

In the previous chapter, we created and styled a few basic routes for Bibliotech's React application so we're ready to begin querying data from the GraphQL API to populate these pages with data. If you don't have a lot of book data in your `db.json` file yet and would like to have some ready-made data to query through the API to display in the user interface over the next three chapters, then you can add the contents of the following file to your `db.json` file before proceeding:

<https://gist.github.com/mandiwise/9e123954246de7d3be8f9fb75663409b>

So far, we queried data from a GraphQL API using a CURL command and a GraphQL IDE, but we're going to need to take a different approach to fetch and render the data in a React component. To do this, we'll use Apollo Client to send GraphQL operations and then cache the data returned from the Bibliotech API. Let's install Apollo Client 3 and its `graphql.js` peer dependency in `client` now:

```
npm i @apollo/client@3.3.11 graphql@15.5.0
```

We'll also add a `.env` file to the `client` directory and set a variable for the GraphQL API endpoint in it. We can use a `.env` file with Create React App as long as we prefix the variables inside it with `REACT_APP_` as follows:

`client/.env`

```
REACT_APP_GRAPHQL_ENDPOINT=http://localhost:4000/graphql
```

Whenever we add or update an environment variable in our React application we'll need to restart it. Additionally, the `NODE_ENV` variable is set automatically for us by Create React App, so we don't need to worry about defining it in this file.

It's important to note that any environment variables we reference from the client-specific `.env` file will be publicly exposed in the browser, so we won't want to put any sensitive information in there (such as API secrets).

Next, we'll create a `graphql` directory in `client/src` to store the Apollo client configuration file. We'll add some operation documents and fragments in this directory later on too. For now, we'll just create an `apollo.js` file in it and add this code:

`client/src/graphql/apollo.js`

```
import { ApolloClient, HttpLink, InMemoryCache } from "@apollo/client";

const client = new ApolloClient({
  cache: new InMemoryCache(),
  connectToDevTools: process.env.NODE_ENV === "development",
  link: new HttpLink({ uri: process.env.REACT_APP_GRAPHQL_ENDPOINT })
});

export default client;
```

Above, the `ApolloClient` constructor takes two important options: an `inMemoryCache` object and an `HttpLink` object that we pass our `REACT_APP_GRAPHQL_API_URL` environment variable into so Apollo Client knows where to send requests. We use an `HttpLink` object instead of providing the GraphQL API endpoint directly to Apollo Client because we can chain together multiple link objects here to customize how data is sent from Apollo Client to the Bibliotech API if desired (we'll learn more about the [Apollo Link API](#) in Chapter 10). We can also optionally set the `connectToDevTools` option to `true` to use the [Apollo Client Dev Tools](#). At the bottom of the file, the default export is the newly instantiated Apollo Client.

Next, we'll import the `ApolloProvider` component into our top-level `index.js` and wrap it around our `Router` component so that all of the pages will be able to use Apollo Client to query data. We'll also need to import the Apollo Client instance we just exported from `apollo.js` and pass it into the `ApolloProvider` component as a prop:

`client/src/index.js`

```
import { ApolloProvider } from "@apollo/client";
// ...

import { Routes } from "./router";
import client from "./graphql/apollo";

function App() {
  return (
    <ApolloProvider client={client}>
      <Router>
        <Routes />
      </Router>
    </ApolloProvider>
  );
}

// ...
```

That's all we need to do to get up and running with Apollo Client! When the development server restarts, the application should look exactly as it did before.

## Add Books to the Index Page

It's time at last to write a query to fetch data from Bibliotech's GraphQL API to display in the React application. Our first query will get a list of books and display them on the index page. To begin, we'll create a `queries.js` file to store all of our query documents, and then write a `GetBooks` query just as we did in Explorer. We'll also wrap the query in a `gql` template tag as we did for the type definitions in the `typeDefs.js` file on the server side:

`client/src/graphql/queries.js`

```
import { gql } from "@apollo/client";

export const GetBooks = gql`  
query GetBooks($limit: Int, $page: Int) {
```

```
books(limit: $limit, orderBy: TITLE_ASC, page: $page) {  
  results {  
    authors {  
      id  
      name  
    }  
    cover  
    id  
    title  
  }  
  pageInfo {  
    hasNextPage  
    page  
  }  
}  
};
```

Before we use this query in the `Index` page component, we'll need to add a couple of new components first. While waiting for data to be fetched asynchronously from the GraphQL API, we should provide feedback in the user interface about that loading state, so we'll use some Tailwind classes to create a CSS-based loader icon with animation in the `index.css` file:

`client/src/index.css`

```
/* ... */  
  
.loader {  
  @apply inline-block h-12 w-12;  
}  
  
.loader:after {  
  @apply block border-4 border-solid rounded-full h-12 w-12;  
  animation: spin 1.2s linear infinite;  
  border-color: #cbd5e0 transparent #cbd5e0 transparent;  
  content: " ";  
}  
  
@keyframes spin {  
  0% {  
    transform: rotate(0deg);  
  }  
  100% {
```

```
        transform: rotate(360deg);
    }
}

@tailwind components;
@tailwind utilities;
```

Now we'll create a reusable Loader component that uses the new `loader` class in it:

*client/src/components/Loader/index.js*

```
function Loader({ centered }) {
  return (
    <div
      className={centered ? "flex-1 flex flex-col items-center justify-center" : ""}>
      <div className="loader" />
    </div>
  );
}

export default Loader;
```

Once the books have been fetched, we'll need some way to display them so the user can browse the list of results. For that, we'll add a BookGrid component:

*client/src/components/BookGrid/index.js*

```
function BookGrid({ books }) {
  return (
    <ul className="gap-8 grid grid-cols-1 sm:grid-cols-2 md:grid-cols-3 lg:grid-cols-4">
      {books.map(({ authors, cover, id, title }) => (
        <li
          className="bg-white cursor-pointer flex flex-col justify-end p-4 shadow-xl"
          key={id}
        >
          <div className="flex flex-1 items-center justify-center">
            {cover ? (
              <img src={cover} alt={`${title} cover`} />
            ) : (
              <div>
                {title}
              </div>
            )}
          </div>
        </li>
      ))}
    </ul>
  );
}

export default BookGrid;
```

```
    ) : (
      <div className="bg-gray-100 border border-solid border-gray-200 flex h-full justify-center items-center mt-4 py-4 px-2 w-full">
        <span className="italic text-center text-gray-600">
          Cover image unavailable
        </span>
      </div>
    )}
</div>
<p className="font-bold leading-tight mt-4 mb-2 hover:text-red-500">
  {title}
</p>
<p className="leading-tight text-gray-600 text-sm">
  {`by ${authors.map(author => author.name).join(", ")}`}
</p>
</li>
))})
</ul>
);
}

export default BookGrid;
```

Lastly, we should display a notice if an error occurs while fetching the book data, so we'll add a reusable `PageNotice` component as well:

`client/src/components/PageNotice/index.js`

```
function PageNotice({ text }) {
  return (
    <div className="flex-auto flex flex-col h-full justify-center text-center">
      <p className="text-gray-500 text-2xl">{text}</p>
    </div>
  );
}

export default PageNotice;
```

Now we can import all of the required components into the `Index` page component, along with the `GetBooks` query and the `useQuery` hook from Apollo Client:

*client/src/pages/Index/index.js*

```
import { useQuery } from "@apollo/client";

import { GetBooks } from "../../graphql/queries";
import BookGrid from "../../components/BookGrid";
import Button from "../../components/Button";
import Loader from "../../components/Loader";
import MainLayout from "../../components/MainLayout";
import PageNotice from "../../components/PageNotice";

// ...
```

Apollo Client 3 provides a hooks-based API to send GraphQL requests, and `useQuery` is the hook that we'll use for all query root operations. At the top of the `Index` component, we'll call the `useQuery` hook and pass in the `GetBooks` query as the first argument followed by a second object argument with a single `variables` key to capture the values we want to send along for the `$limit` and `$page` variables (previously, we set these values using the Variables panel of Explorer):

*client/src/pages/Index/index.js*

```
// ...

function Index() {
  const { data, error, loading } = useQuery(GetBooks, {
    variables: { limit: 12, page: 1 }
  });

  // ...
}

export default Index;
```

Next, we'll update what's get returned from this component and do some conditional rendering based on the state of the query when it's either `loading`, or we received some `data` from the API that we can display, or an `error` occurred at some point in the request:

*client/src/pages/Index/index.js*

```
// ...

function Index() {
```

```
// ...

let content = null;

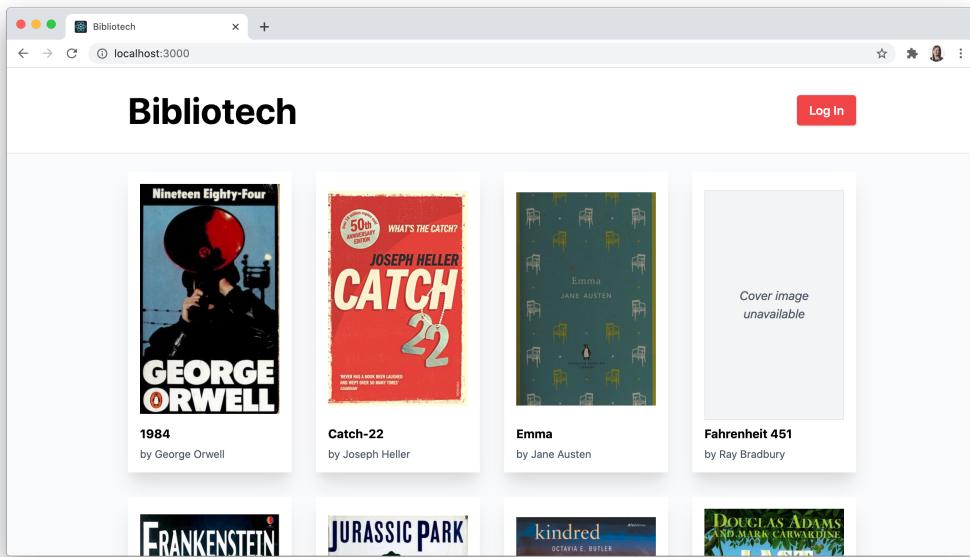
if (loading && !data) {
  content = <Loader centered />;
} else if (data?.books) {
  const {
    pageInfo: { hasNextPage },
    results
  } = data.books;

  content = (
    <>
      <div className="mb-8">
        <BookGrid books={results} />
      </div>
      {hasNextPage && (
        <div className="flex justify-center">
          <Button text="Load More" type="button" />
        </div>
      )}
    </>
  );
} else if (error) {
  content = <PageNotice text="Book list is unavailable. Please try again." />;
}

return <MainLayout>{content}</MainLayout>;
}

export default Index;
```

With this code in place, the index page will now render a book list with the first 12 results displayed in a grid:



For now, the “Load More” button will be a placeholder, but we’ll update this component to fetch additional pages of results in Chapter 9 (where we’ll also use the page field fetched with the query).

## Send the JWT in a Cookie

Now that we have Apollo Client configured and we can successfully query data from a GraphQL API, the balance of this chapter will focus primarily on handling authentication on the client side. We previously added `login` and `signUp` mutations that create a signed JWT to send back in the response so that a client can then include that JWT with subsequent requests that require authentication. When using a GraphQL IDE to test an API, we had to add that token manually as the request’s `Authorization` header, and that value was persisted in the interface until we removed it or refreshed the page. In a React application, we’ll need to take a different approach to persist the returned token.

Storing JWTs in local storage is a dangerous practice and would leave Bibliotech exposed to cross-site scripting (XSS) attacks.<sup>1</sup> Many tutorials use local storage to persist JWTs because it’s quick and easy. These tutorials sometimes call out that you wouldn’t want to do this in a real production application, but it would be instructional here if we lay a foundation for handling JWTs in a more secure way. That leaves us with two options:

<sup>1</sup><https://www.rdegges.com/2018/please-stop-using-local-storage/>

1. Persist the token **in-memory** (and specifically for our purposes, in React state) and then retrieve the token value from memory to include as an `Authorization` header with each request sent from Apollo Client
2. Set the token as an **HttpOnly cookie** and include the cookie with each request sent to the API from Apollo Client

Ideally, we would implement the first option because this would be the securest approach, but a basic implementation of in-memory persistence has a serious downside in that a user would need to re-authenticate every time they reload the page. To avoid this, we would need to engineer a more advanced solution that uses refresh tokens<sup>2</sup> to re-authenticate a user automatically between page loads if their session is still valid. Many third-party authentication services take this approach in their client SDKs, further highlighting that it's worthwhile to investigate using such a service in a real application. However, rolling our own refresh token system is outside the scope of this book.

So given these practical considerations, that leaves us with the second option to use an `HttpOnly` cookie. Using an `HttpOnly` cookie means that the cookie will only be sent with HTTP requests and won't be accessible to any scripts running in the browser. This is an important safety feature but it's important to note that it won't completely protect us from more advanced XSS attacks, but a Content Security Policy (CSP)<sup>3</sup> can provide some additional protection against this. Cookies are also vulnerable to cross-site request forgery (CSRF) attacks, but creating and including an anti-CSRF token with requests would help mitigate this risk. Additionally, in production, it would also be a good idea to set the "Secure" attribute on the cookie so that it will only be sent with encrypted requests using the HTTPS protocol and also configuring the "SameSite" attribute to be as strict as possible (by default, the `SameSite` attribute's value will be "Lax" for cookies that don't have the `Secure` attribute set).

It may seem that there are many potential pitfalls when using cookies and that there would be considerable overhead to use them safely in production (both of these observations are correct!). That said, choosing this option will at least provide a middle ground between declaring open season on our JWTs by keeping them in local storage and spending several more chapters building out an in-memory, client-side authentication system that can handle fetching refresh tokens between page loads. For brevity's sake, we won't be augmenting the cookie implementation in our development environment with a CSP, anti-CSRF tokens, or configuring the `Secure` or `SameSite` attributes. Again, these concerns are outside of the scope of this GraphQL-focused book, so please consider doing further research in these topics. But the code that we do write in the sections that follow will provide an initial foundation for hardening how the JWT is handled in a real production environment.

Alright, the security public service announcement is over now so let's get back to the task at hand. As our first order of business, we need to make some updates to the server. When a user logs in or signs up we'll need to capture the value of the token that's generated in these mutations and add it to the `Set-Cookie` header in the response. This will pose a bit of a challenge given our current

---

<sup>2</sup>[https://hasura.io/blog/best-practices-of-using-jwt-with-graphql/#refresh\\_token](https://hasura.io/blog/best-practices-of-using-jwt-with-graphql/#refresh_token)

<sup>3</sup><https://developer.mozilla.org/en-US/docs/Web/HTTP/CSP>

set-up because mutations don't have access to the Express `res` object, which is where we'll need to set the cookie for it to be delivered with the response. There are a couple of different ways we can solve this problem. The first would involve adding the entire `res` object directly to the Apollo Server context so that it's accessible in every resolver, but this seems a bit heavy-handed. Another more targeted approach would be to use a custom [Apollo Server plugin](#).

The Apollo Server plugin API allows us to respond to events at various points in the lifecycle of a request to a GraphQL API by leveraging a series of hooks. Specifically, we'll use the `willSendResponse` hook to intercept the output of a `login` or `signUp` mutation at the point immediately before the response is delivered to the client. We will then get the value from the `token` field in that output to use as the cookie value and attach the cookie to the response. Before creating the plugin, we'll need to install a new library to assist with serializing the cookie data with its attributes. We'll jump back over to the `server` directory and install this package now:

```
npm i cookie@0.4.1
```

Next, we'll create a new directory called `plugins` in `server/src/graphql` and add a `cookieHeaderPlugin.js` file to it with the following code:

`server/src/graphql/plugins/cookieHeaderPlugin.js`

```
import cookie from "cookie";

const cookieHeaderPlugin = {
  requestDidStart() {
    return {
      willSendResponse({ operation, response }) {
        // Do something to the response before sending it here...
      }
    };
  }
};

export default cookieHeaderPlugin;
```

An Apollo Server plugin can tap into two high-level events: `serverWillStart` for server-related events and `requestDidStart` for request-related events. We'll choose `requestDidStart` here, and then return an object containing the request-specific `willSendResponse` method where we will execute the code that intercepts the `login` and `signUp` mutations.

The `willSendResponse` method has access to a single `requestContext` parameter from which we can destructure the `operation` object (so we can check if we're dealing with the right kind of mutation) and the `response` (so we can get the `token` and set the cookie). After verifying that the operation is a mutation, we'll have to look inside the field selection set to see if it's a `login` or

`signUp` mutation and return from the function if it's not. Otherwise, we'll proceed with getting the `token` field value in the response and adding it as an `HttpOnly` cookie named `token`:

`server/src/graphql/plugins/cookieHeaderPlugin.js`

```
import cookie from "cookie";

const cookieHeaderPlugin = {
  requestDidStart() {
    return {
      willSendResponse({ operation, response }) {
        if (operation?.operation === "mutation") {
          const authMutation = operation.selectionSet.selections.find(
            selection =>
              selection.name.value === "login" ||
              selection.name.value === "signUp"
          );

          if (!authMutation) {
            return;
          }

          const fieldName = authMutation.name.value;

          if (response.data?.[fieldName].token) {
            const cookieString = cookie.serialize(
              "token",
              response.data[fieldName].token,
              { httpOnly: true, maxAge: 86400 }
            );
            response.http.headers.set("Set-Cookie", cookieString);
          }
        }
      };
    };
  };
};

export default cookieHeaderPlugin;
```

Note that the `maxAge` of the cookie is equivalent to one day in seconds to match the lifespan of the signed JWT that was created in the mutation. The long lifespan of the token and cookie will make things easier during development, but it would be more secure to use a shorter lifespan and refresh the token behind the scenes while a user's session is valid.

To add the new plugin, we'll need to update the top-level `index.js` file on the server to include a `plugins` option in `ApolloServer` and then add the `cookieHeaderPlugin` to an array as the option's value (we use an array because we can add multiple plugins if needed):

`server/src/index.js`

```
// ...

import cookieHeaderPlugin from "./graphql/plugins/cookieHeaderPlugin.js";
// ...

const server = new ApolloServer({
  // ...
  context: ({ req }) => {
    const user = req.user || null;
    return { user };
  },
  plugins: [cookieHeaderPlugin]
});

// ...
```

Using a cookie, we also need to do some extra configuration to the Express JWT middleware. By default, this middleware will look for a JWT in the `Authorization` header. However, we now need the middleware to check this header or the `token` cookie. To keep things organized, let's create a new `tokens.js` file in `server/src/utils` that contains this code:

`server/src/utils/tokens.js`

```
export function getToken(req) {
  if (
    req.headers.authorization &&
    req.headers.authorization.split(" ")[0] === "Bearer"
  ) {
    return req.headers.authorization.split(" ")[1];
  } else if (req.headers.cookie) {
```

```
const tokenCookie = req.headers.cookie
  .split("; ")
  .find(cookie => cookie.includes("token"));

  return tokenCookie ? tokenCookie.split("=")[1] : null;
}
return null;
}
```

Let's also move the callback that handles invalid tokens in this middleware into a dedicated function in the `tokens.js` file as well now:

`server/src/utils/tokens.js`

```
// ...

export function handleInvalidToken(err, req, res, next) {
  if (err.code === "invalid_token") {
    return next();
  }
  return next(err);
}
```

Now can import the two functions and update the `expressJwt` middleware to use them:

`server/src/index.js`

```
// ...

import { getToken, handleInvalidToken } from "./utils/tokens.js";
// ...

app.use(
  expressJwt({
    secret: process.env.JWT_SECRET,
    algorithms: ["HS256"],
    credentialsRequired: false,
    getToken
  }),
  handleInvalidToken
);
```

```
// ...
```

Before jumping back to the client, there's one more thing we need to consider: what happens when a user logs out of the application? For our purposes, logging a user out of Bibliotech will mean invalidating their HttpOnly cookie, and this can only be done on the server side. To handle logout requests from the client, we'll add a `logout` mutation to API:

*server/src/graphql/typeDefs.js*

```
import { gql } from "apollo-server-express";

const typeDefs = gql`  
# ...  
  
type Mutation {  
# ...  
  "Logs an authenticated user out."  
  logout: Boolean!  
# ...  
}  
;  
  
export default typeDefs;
```

The `logout` resolver will simply return `true` because cookie invalidation will need to happen in the Apollo Server plugin we previously created:

*server/src/graphql/resolvers.js*

```
// ...  
  
const resolvers = {  
// ...  
Mutation: {  
// ...  
  logout(root, args, context, info) {  
    return true;  
  },  
// ...  
};  
};
```

```
export default resolvers;
```

Now we'll update the `cookieHeaderPlugin` to intercept any `logout` mutation too and invalidate the `token` cookie by creating a new cookie with an expiration date in the past:

`server/src/graphql/plugins/cookieHeaderPlugin.js`

```
import cookie from "cookie";

const cookieHeaderPlugin = {
  requestDidStart() {
    return {
      willSendResponse({ operation, response }) {
        if (operation.operation === "mutation") {
          const authMutation = operation.selectionSet.selections.find(
            selection =>
              selection.name.value === "login" ||
              selection.name.value === "logout" ||
              selection.name.value === "signUp"
          );

        if (!authMutation) {
          return;
        }

        const fieldName = authMutation.name.value;

        if (fieldName === "logout") {
          const cookieString = cookie.serialize("token", "", {
            httpOnly: true,
            expires: new Date(1)
          });
          response.http.headers.set("Set-Cookie", cookieString);
        } else {
          if (response.data?.[fieldName].token) {
            const cookieString = cookie.serialize(
              "token",
              response.data[fieldName].token,
              { httpOnly: true, maxAge: 86400 }
            );
            response.http.headers.set("Set-Cookie", cookieString);
          }
        }
      }
    }
  }
}
```

```
        }
      }
    }
  };
}

export default cookieHeaderPlugin;
```

With this new and improved server code in place, we're now fully equipped to issue and handle the `token` cookie, so we're ready to jump back to the client side.

## Log In or Register a New User from the Client

For our cookie to be set in the browser and included with each subsequent request to our GraphQL API, we need to run the React application on the same port as the GraphQL API. We can easily configure this with Create React App by setting a `proxy` property in the client's package.json file with a value of `http://localhost:4000`:

*client/package.json*

```
{
  "name": "client",
  "version": "0.1.0",
  "proxy": "http://localhost:4000",
  // ...
}
```

We must also update the `REACT_APP_GRAPHQL_ENDPOINT` variable to use port 3000 in the `.env` file:

*client/.env*

```
REACT_APP_GRAPHQL_ENDPOINT=http://localhost:3000/graphql
```

Next, we're going to set up a mutation to log existing users in first and then adapt this approach to support user sign-ups afterward. We'll begin by creating a `mutations.js` file in `client/src/graphql` and adding a `Login` mutation to it:

client/src/graphql/mutations.js

```
import { gql } from "@apollo/client";

export const Login = gql`  
  mutation Login($password: String!, $username: String!) {  
    login(password: $password, username: $username) {  
      viewer {  
        id  
        email  
        name  
        username  
      }  
      token  
    }  
  }  
`;
```

In the `Login` page component, we'll import the mutation as `LoginMutation` to avoid a name conflict. We'll also import the `useHistory` hook from React Router so we can push a user to the `/home` route after successfully authenticating. Lastly, we'll need the `useMutation` hook from Apollo Client. The `useMutation` works much like the `useQuery` hook does, but this hook returns an array where the first item is a function that can be used to send the mutation when needed. In the `Login` page component, the event that will trigger the mutation is a form submission.

The second item in the array is an object like the one we destructured directly from the `useQuery` hook previously, and it contains information about the state of the operation. Also note that in addition to passing in the mutation operation into `useMutation`, we also pass an `onCompleted` option to redirect the user to the homepage after a successful mutation. To send the value of the `username` and `password` inputs as mutation variables, we must also convert them to **controlled components** using the `useState` hook:

client/src/components/Login/index.js

```
import { useHistory } from "react-router-dom";  
import { useMutation } from "@apollo/client";  
import { useState } from "react";  
  
import { Login as LoginMutation } from "../../graphql/mutations";  
// ...  
  
function Login() {  
  const [isMember, setIsMember] = useState(true);
```

```
const [password, setPassword] = useState("");
const [username, setUsername] = useState("");
const history = useHistory();

const onCompleted = () => {
  history.push("/home");
};

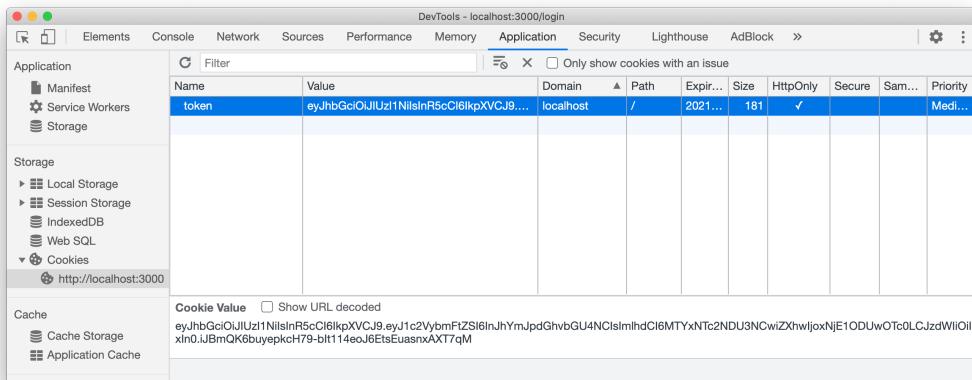
const [login, { error, loading }] = useMutation(LoginMutation, {
  onCompleted
});

return (
  <div className="bg-gray-50 flex items-center justify-center min-h-screen">
    <div className="bg-white shadow p-8 max-w-sm w-10/12">
      <h1 className="mb-4 text-2xl">Welcome to Bibliotech</h1>
      <form
        onSubmit={async event => {
          event.preventDefault();
          await login({ variables: { password, username } }).catch(err =>
            {
              console.error(err);
            });
        }}
      >
        <TextInput
          className="mb-4"
          hiddenLabel
          id="username"
          inputWidthClass="w-full"
          label="Username"
          onChange={event => {
            setUsername(event.target.value);
          }}
          placeholder="Your username"
          type="text"
          value={username}
        />
        {/* ... */}
        <TextInput
          className="mb-8"
```

```
    hiddenLabel
    id="password"
    inputWidthClass="w-full"
    label="Password"
    onChange={event => {
      setPassword(event.target.value);
    }}
    placeholder="Your password"
    type="password"
    value={password}
  />
  <div className="flex items-center">
    <Button
      className="mr-2"
      disabled={loading}
      text={isMember ? "Log In" : "Sign Up"}
      type="submit"
    />
    {/* ... */}
  </div>
  {error && (
    <p className="mt-4 text-red-500 text-sm">{error.message}</p>
  )}
</form>
</div>
</div>
);
}

export default Login;
```

If we log in an existing user through the React application and then open the browser's developer tools, we should see an `HttpOnly` token cookie stored with the JWT value in it:



Now we'll wire up the sign-up version of the form. First, we'll need a new mutation:

`client/src/graphql/mutations.js`

```
import { gql } from "@apollo/client";

// ...

export const SignUp = gql`mutation SignUp($input: SignUpInput!) {
  signUp(input: $input) {
    viewer {
      id
      email
      name
      username
    }
    token
  }
}`;
```

But wait! The code we just added to `mutations.js` doesn't seem very DRY because both the `Login` and `SignUp` mutations use the same field selections. This seems like a good opportunity to add a reusable fragment (as we did in Chapter 2). Let's add a `fragments.js` file to `client/src/graphql` and add the following code to it:

client/src/graphql/fragments.js

```
import { gql } from "@apollo/client";

export const viewerAndToken = gql`  
  fragment viewerAndToken on AuthPayload {  
    viewer {  
      id  
      email  
      name  
      username  
    }  
    token  
  }  
`;
```

Now we can refactor our two mutations to use the `viewerAndToken` fragment. For each operation where the fragment is used, we must include the `viewerAndToken` variable just after the closing curly bracket at the end of the template string:

client/src/graphql/mutations.js

```
import { gql } from "@apollo/client";

import { viewerAndToken } from "./fragments";

export const Login = gql`  
  mutation Login($password: String!, $username: String!) {  
    login(password: $password, username: $username) {  
      ...viewerAndToken  
    }  
  }  
  ${viewerAndToken}  
`;  
  
export const SignUp = gql`  
  mutation SignUp($input: SignUpInput!) {  
    signUp(input: $input) {  
      ...viewerAndToken  
    }  
  }  
  ${viewerAndToken}  
`;
```

Lastly, we'll import the `SignUp` mutation into the `Login` page component and call `useMutation` again to set up the new mutation. We'll need to update the form's `onSubmit` prop to call either mutation conditionally depending on the `isMember` state value. We must also turn the name and email inputs into controlled components as well now:

`client/src/components/Login/index.js`

```
// ...

import { Login as LoginMutation, SignUp } from "../../graphql/mutations";
// ...

function Login() {
  const [email, setEmail] = useState("");
  const [isMember, setIsMember] = useState(true);
  const [name, setName] = useState("");
  // ...

  const [login, { error, loading }] = useMutation(LoginMutation, {
    onCompleted
  });
  const [signUp] = useMutation(SignUp, { onCompleted });

  return (
    <div className="bg-gray-50 flex items-center justify-center min-h-screen">
      <div className="bg-white shadow p-8 max-w-sm w-10/12">
        <h1 className="mb-4 text-2xl">Welcome to Bibliotech</h1>
        <form
          onSubmit={async event => {
            event.preventDefault();

            if (isMember) {
              await login({ variables: { password, username } }).catch(err
                => {
                  console.error(err);
                });
            } else {
              await signUp({
                variables: { input: { email, name, password, username } }
              }).catch(err => {
                console.error(err);
              });
            };
          }}
        </form>
      </div>
    </div>
  );
}
```

```
        }
    }()
>
 {/* ... */}
{!isMember && (
    <>
        <TextInput
            className="mb-4"
            hiddenLabel
            id="email"
            inputWidthClass="w-full"
            label="Email"
            onChange={event => {
                setEmail(event.target.value);
            }}
            placeholder="Your email"
            type="email"
            value={email}
        />
        <TextInput
            className="mb-4"
            hiddenLabel
            id="name"
            inputWidthClass="w-full"
            label="Full Name"
            onChange={event => {
                setName(event.target.value);
            }}
            placeholder="Your full name"
            type="text"
            value={name}
        />
    </>
)}
```

Try signing up a new user through the form now. You should see the new user appear in `db.json` and a cookie stored in the browser after the mutation completes.

## Use Context and Hooks to Manage Authentication State

Now that we can store the cookie in the browser, it would be helpful if we could persist the viewer's data provided by the `AuthPayload` and also keep track of whether their session is still valid (where validity is based on the token's expiration time). To do this, we'll keep track of the viewer's personal information in memory only but we'll persist the token expiration time to local storage so we can check if a user still has a valid session after a page reload. If the page reloads and the token is still fresh when checked, the GraphQL API will be queried to refetch the viewer's data again, and with the `HttpOnly` cookie in tow.

It will ultimately prove useful to have access to the viewer's data throughout various components in the client application, so we'll use React's context API for this. Context is a handy way to pass data around a React application without passing it through props at each level in the component tree. Context is useful when there is some kind of global data that we want to share across many React components. In our case, that data is information about the authenticated user.

Creating a context in React consists of three parts:

- A **context object** that holds current values for the context
- A **provider** component that allows other components to “consume” (in other words, subscribe to) the context changes
- A **consumer** component or the much cleaner `useContext` hook (available in React 16.8+) that subscribes to context changes and can be used to inform other components nested inside that they need to update

To create a context to manage authentication state, we'll begin by adding a new `context` directory in `client/src` and then add an `AuthContext` directory to that with an `index.js` file:

`client/src/context/AuthContext/index.js`

```
import { createContext, useContext, useState } from "react";

const AuthContext = createContext();
const useAuth = () => useContext(AuthContext);

function AuthProvider({ children }) {
  const [checkingSession, setCheckingSession] = useState(true);

  // Token expiration and viewer-handling logic will go here...

  return (
    <AuthContext.Provider value={{ checkingSession, setCheckingSession }}>
      {children}
    </AuthContext.Provider>
  );
}

export default AuthProvider;
```

```
<AuthContext.Provider value={{ checkingSession }}>
  {children}
</AuthContext.Provider>
);
}

export { AuthProvider, useAuth };
```

In this file, we first create an `AuthContext` and then create a `useAuth` wrapper function that returns the result of calling the `useContext` hook with the `AuthContext` object passed into it. The return value of `useAuth` will ultimately be the current value for the `AuthContext`. In other words, if we create a component and call `useAuth` inside of it, the return value of `useAuth` will be the object that is passed into the `value` prop of the `AuthContext.Provider` component. For now, that object would only contain a `checkingSession` property with a value of `true`.

Next, we'll use the `AuthProvider` component at the high level in our component tree so other components will have access to the `AuthContext` wherever needed. Let's head over to `client/src/index.js` and nest the `AuthProvider` just inside of the `ApolloProvider` component:

`client/src/index.js`

```
// ...

import { AuthProvider } from "./context/AuthContext";
// ...

function App() {
  return (
    <ApolloProvider client={client}>
      <AuthProvider>
        <Router>
          <Routes />
        </Router>
      </AuthProvider>
    </ApolloProvider>
  );
}

// ...
```

We have to put the `AuthProvider` inside the `ApolloProvider` because the inside of the `AuthProvider` component we will need access to the instance of Apollo Client to make a query to get the current viewer's data when the page reloads.

To persist the token expiration time in local storage, we'll need to decode the token received in the response from a `login` or `signUp` mutation to retrieve the expiration time. We can install the `jwt-decode` package in the `client` directory to assist with that:

```
npm i jwt-decode@3.1.2
```

### Schema Design Best Practice Alert!

This package only decodes the JWT's base64Url encoding and it does not validate the JWT. To validate the JWT, we would need access to the secret that was used to sign it and we wouldn't want to expose that to the client.

We can trust our server will send us a valid JWT for our development purposes here, but this concern raises a very important question: *should the client have to fuss over doing the work to decode the JWT to get the token expiration in the first place, or should the expiration time be codified in the GraphQL schema instead?*

As an extra credit project—and with a mindset of taking an iterative, client-focused approach to API design—you may wish to try adding a `tokenExpiration` field to the `AuthPayload` and calculate this value on the server instead.

Next, we'll import the new package into the `AuthContext` component and create a `persistSessionData` function that gets the JWT expiration time from the decoded token, converts it to milliseconds, saves it to local storage, and then stores the `viewer` data in state too. We'll also add an `isAuthenticated` function to compare this value against the expiration time saved in local storage, and then expose `persistSessionData`, `isAuthenticated`, and the `viewer` values via the context:

`client/src/context/AuthContext/index.js`

```
import { createContext, useContext, useState } from "react";
import jwtDecode from "jwt-decode";

// ...

function AuthProvider({ children }) {
  const [checkingSession, setCheckingSession] = useState(true);
  const [viewer, setViewer] = useState();
```

```
const persistSessionData = authPayload => {
  if (authPayload.token && authPayload.viewer) {
    const decodedToken = jwtDecode(authPayload.token);
    const expiresAt = decodedToken.exp * 1000;
    localStorage.setItem("token_expires_at", expiresAt);
    setViewer(authPayload.viewer);
  }
};

const isAuthenticated = () => {
  const expiresAt = localStorage.getItem("token_expires_at");
  return expiresAt ? new Date().getTime() < expiresAt : false;
};

return (
  <AuthContext.Provider
    value={{
      checkingSession,
      isAuthenticated,
      persistSessionData,
      viewer
    }}
  >
  {children}
  </AuthContext.Provider>
);
}

export { AuthProvider, useAuth };
```

The `persistSessionData` function will be called after a successful `login` or `signUp` mutation runs and then set the token expiration time and the `viewer` based on the value of those fields in the returned `AuthPayload`. However, if the page reloads at a time when `isAuthenticated` still returns `true`, then we'll need a way to fetch the `viewer` again when the `AuthProvider` component mounts. We'll need a `GetViewer` query to do that:

`client/src/graphql/queries.js`

```
// ...

export const GetViewer = gql`  
  query GetViewer {
```

```
viewer {
  id
  email
  name
  username
}
};
```

Inside of the `AuthProvider` component, we'll take a slightly different approach in how we use this query to get the viewer. Instead of using the `useQuery` hook to automatically fetch the data when the component loads, we'll instead use the `useApolloClient` hook to access the Apollo Client instance directly and send a query with its `query` method when the user has a valid token but the `viewer` data isn't available.

To run this check and fetch the data when the `AuthProvider` loads, we'll use React's `useEffect` hook because it behaves much like the `componentDidMount` lifecycle method did in a traditional class component. Essentially, we use this hook whenever we want to do something that will have a side effect in our application—in this case, that side effect is fetching data. The `useEffect` hook expects us to pass it a function. Inside this function, we call `client.query` and pass in the `GetViewer` operation:

`client/src/context/AuthContext/index.js`

```
import { useApolloClient } from "@apollo/client";
import { createContext, useContext, useEffect, useState } from "react";
import jwtDecode from "jwt-decode";

import { GetViewer } from "../../graphql/queries";

const AuthContext = createContext();
const useAuth = () => useContext(AuthContext);

function AuthProvider({ children }) {
  const [checkingSession, setCheckingSession] = useState(true);
  const [error, setError] = useState();
  const [viewer, setViewer] = useState();
  const client = useApolloClient();

  // ...

  useEffect(() => {
    const getViewerIfAuthenticated = async () => {
      try {
        if (localStorage.getItem("token")) {
          const decodedToken = jwtDecode(localStorage.getItem("token"));
          if (decodedToken.exp * 1000 < Date.now()) {
            setCheckingSession(false);
            setError("Session has expired");
          } else {
            const { data } = await client.query({
              query: GetViewer,
            });
            setViewer(data.viewer);
            setCheckingSession(false);
          }
        }
      } catch (err) {
        setError(err.message);
      }
    };
    getViewerIfAuthenticated();
  }, []);
}

AuthContext.Provider = AuthProvider;
```

```
    if (isAuthenticated() && !viewer) {
      try {
        const { data, error: viewerError, loading } = await
          client.query({
            query: GetViewer
          });

        if (!loading && viewerError) {
          setError(viewerError);
        } else if (!loading && data) {
          setViewer(data.viewer);
        }
      } catch (err) {
        setError(err);
      }
    }
    setCheckingSession(false);
  };
  getViewerIfAuthenticated();
}, [client, viewer]);

return (
  <AuthContext.Provider
    value={{
      checkingSession,
      error,
      isAuthenticated,
      persistSessionData,
      viewer
    }}
  >
  {children}
  </AuthContext.Provider>
);
}

export { AuthProvider, useAuth };
```

We create an `async getViewerIfAuthenticated` function inside of `useEffect` and immediately call it because `useEffect` can only return another (optional) function to do clean-up when the component is unmounted. If we were to mark the `useEffect` callback as `async` directly to use the `await` keyword with `client.query`, then an error would result because `async` functions always return promises.

With this code in place, we can now use our `useAuth` hook in the `Login` component to persist the viewer data and token expiration time based on the `signUp` or `login` mutation response:

*client/src/components/Login/index.js*

```
// ...
import { useAuth } from "../../context/AuthContext";
// ...

function Login() {
  // ...
  const { persistSessionData } = useAuth();
  const history = useHistory();

  const onCompleted = data => {
    const { token, viewer } = Object.entries(data)[0][1];
    persistSessionData({ token, viewer });
    history.push("/home");
  };

  // ...
}

export default Login;
```

Try to log in a user again and use the browser's developer tools to check local storage afterward and confirm the expiration time was successfully stored.

## Log Out a User

We now need to handle logouts on the client side as well. We'll begin by adding a logout mutation to `mutations.js`:

*client/src/graphql/mutations.js*

```
// ...

export const Logout = gql` 
  mutation Logout {
    logout
  }
`;
```

```
// ...
```

Logging out on the client side will mean sending the `Logout` mutation to trigger the server to invalidate the cookie, but we will also need to do some clean-up to remove the `viewer` object from state and the token expiration time from local storage. To do that, we'll add a `clearSessionData` function to the `AuthProvider`:

`client/src/context/AuthContext/index.js`

```
// ...

function AuthProvider({ children }) {
  // ...

  const clearSessionData = () => {
    localStorage.removeItem("token_expires_at");
    setViewer(null);
  };

  // ...

  return (
    <AuthContext.Provider
      value={{
        checkingSession,
        clearSessionData,
        // ...
      }}
    >
      {children}
    </AuthContext.Provider>
  );
}

export { AuthProvider, useAuth };
```

Next, we'll update the `NavBar` component for authenticated users so that it conditionally renders the button in the righthand corner to display "Log Out," run the `Logout` mutation when clicked, clear the session data, and push the user to the `/login` route after the mutation completes. While we're working on the `NavBar` component, let's also add a "My Library" link to allow authenticated users to easily navigate to the homepage:

client/src/components/NavBar/index.js

```
import { Link, useHistory } from "react-router-dom";
import { useMutation } from "@apollo/client";

import { Logout } from "../../graphql/mutations";
import { useAuth } from "../../context/AuthContext";
import Button from "../Button";

function NavBar() {
  const { clearSessionData, isAuthenticated } = useAuth();
  const history = useHistory();

  const [logout] = useMutation(Logout, {
    onCompleted: () => {
      clearSessionData();
      history.push("/login");
    }
  });

  return (
    <header className="bg-white border-b border-gray-200 border-solid">
      <div className="flex flex-wrap items-center justify-between mx-auto max-w-screen-lg px-8 py-8 w-full">
        {/* ... */}
        <div className="flex items-center sm:justify-end mt-2 sm:mt-0">
          {isAuthenticated() && (
            <Link to="/home">
              <span className="font-semibold mr-4 text-sm sm:text-base text-red-600">
                My Library
              </span>
            </Link>
          )}
          <Button
            onClick={event => {
              event.preventDefault();

              if (isAuthenticated()) {
                logout();
              } else {
                history.push("/login");
              }
            }}
          >{isAuthenticated() ? "Logout" : "Login"}</Button>
        </div>
      </div>
    </header>
  );
}
```

```
        }
    //}
    text={isAuthenticated() ? "Log out" : "Log In"}
  />
</div>
</div>
</header>
);
}

export default NavBar;
```

Before moving on, try going through the entire authentication flow for a user now and confirm that they are successfully logged out at the end.

## Render Private and Public Routes

We can finally log users in and out of the Bibliotech React application, so the next logical step is to prevent unauthenticated users from navigating to routes that should only be accessible once logged in. Specifically, we're going to make the `/home` route private but leave the `/` and `/login` routes public. Additionally, it wouldn't make sense to allow authenticated users to access the `/login` page, so we'll redirect them to the `/home` route if they try to do this.

We'll begin by creating a `PrivateRoute` component. Add a `PrivateRoute` directory with an `index.js` file in it to `client/src/components` with the following code:

`client/src/components/PrivateRoute/index.js`

```
import { Redirect, Route, useLocation } from "react-router-dom";
import { useEffect } from "react";

import { useAuth } from "../../context/AuthContext";
import Loader from "../../components/Loader";

function PrivateRoute({ component: Component, ...rest }) {
  const { checkingSession, isAuthenticated } = useAuth();
  const { pathname } = useLocation();

  const renderRoute = props => {
    if (checkingSession) {
      return (
        <div className="flex h-screen">
```

```
        <Loader centered />
      </div>
    );
} else if (isAuthenticated()) {
  return <Component {...props} />;
}

return <Redirect to="/login" />;
};

useEffect(() => {
  window.scrollTo(0, 0);
}, [pathname]);

return <Route {...rest} render={renderRoute} />;
}

export default PrivateRoute;
```

This component is a wrapper for React Router's `Route` component, but with some control flow built into it to determine what to display based on the user's authentication state. First, we need to determine if the application is currently checking the session. If it is, then we'll display the `Loader` component and wait for this to finish. If the session is checked and the user is authenticated, then we'll show the component that should be rendered by the route. Otherwise, we'll use React Router's `Redirect` component to send the user back to the index page.

To keep our code tidy, we'll wrap these checks in a function called `renderRoute` inside the `PrivateRoute` component. Below this function, we also take care of `scroll restoration` in the `useEffect` hook so the window scrolls back to the top when the user navigates to another route. We'll finish off this component by passing any remaining route props through by spreading `rest`, and also passing the `renderRoute` function to the `render` prop of `Route`.

We'll need to check if a user is authenticated when rendering a public route too, so we'll add a modified version of the `PrivateRoute` component as a new `PublicRoute` component:

`client/src/components/PublicRoute/index.js`

```
import { Route, useLocation } from "react-router-dom";
import { useEffect } from "react";

import { useAuth } from "../../context/AuthContext";
import Loader from "../../components/Loader";
```

```
function PublicRoute({ component: Component, ...rest }) {
  const { checkingSession } = useAuth();
  const { pathname } = useLocation();

  const renderRoute = props => {
    if (checkingSession) {
      return (
        <div className="flex h-screen">
          <Loader centered />
        </div>
      );
    }

    return <Component {...props} />;
  };

  useEffect(() => {
    window.scrollTo(0, 0);
  }, [pathname]);

  return <Route {...rest} render={renderRoute} />;
}

export default PublicRoute;
```

Now we can update all of our existing routes to use the `PrivateRoute` and `PublicRoute` components instead of the generic `Route` (we can also remove the `Route` import on the first line):

`client/src/router/index.js`

```
import { Switch } from "react-router";

import Index from "../pages/Index";
import Home from "../pages/Home";
import Login from "../pages/Login";
import PrivateRoute from "../components/PrivateRoute";
import PublicRoute from "../components/PublicRoute";

export function Routes() {
  return (
    <Switch>
      <PublicRoute exact path="/" component={Index} />
```

```
        <PrivateRoute exact path="/home" component={Home} />
        <PublicRoute exact path="/login" component={Login} />
    </Switch>
);
}

export default Routes;
```

Lastly, we'll update the `Login` component to check if a user is authenticated and then redirect them to the `/home` route if they are:

`client/src/pages/Login/index.js`

```
import { Redirect, useHistory } from "react-router-dom";
// ...

function Login() {
// ...
const { isAuthenticated, persistSessionData } = useAuth();
const history = useHistory();

// ...

return isAuthenticated() ? (
<Redirect to="/home" />
) : (
<div className="bg-gray-50 flex items-center justify-center min-h-screen">
 {/* ... */}
</div>
);
}

export default Login;
```

Head over to the React application and try accessing the `/home` route before logging in and the `/login` route after authenticating to confirm that the redirection works as expected.

## Display a User's Library on the Home Page

As our final order of business in this chapter, we'll query some content for the homepage. This page will display the same `BookGrid` component as the index page, but only list the books that

a user has added to their library. As a first step, we'll create a new `GetViewerLibrary` query in `queries.js`:

`client/src/graphql/queries.js`

```
// ...  
  
export const GetViewerLibrary = gql`  
query GetViewerLibrary($limit: Int, $page: Int) {  
  viewer {  
    id  
    library(limit: $limit, orderBy: ADDED_ON_DESC, page: $page) {  
      results {  
        authors {  
          id  
          name  
        }  
        cover  
        id  
        title  
      }  
      pageInfo {  
        hasNextPage  
        page  
      }  
    }  
  }  
};`;
```

Again, we see that have an operation with nearly identical fields to another operation (the `GetBooks` query in this case), so it would make sense to create another fragment in `fragments.js`:

`client/src/graphql/fragments.js`

```
import { gql } from "@apollo/client";  
  
export const basicBook = gql`  
fragment basicBook on Book {  
  authors {  
    id  
    name
```

```
    }
    cover
    id
    title
}
`;

// ...
```

Now we'll use the `basicBook` fragment for the `GetBooks` and `GetViewerLibrary` queries:

`client/src/graphql/queries.js`

```
import { gql } from "@apollo/client";

import { basicBook } from "./fragments";

export const GetBooks = gql`  
query GetBooks($limit: Int, $page: Int) {  
  books(limit: $limit, orderBy: TITLE_ASC, page: $page) {  
    results {  
      ...basicBook  
    }  
    pageInfo {  
      hasNextPage  
      page  
    }  
  }  
  ${basicBook}  
};  
  
// ...  
  
export const GetViewerLibrary = gql`  
query GetViewerLibrary($limit: Int, $page: Int) {  
  viewer {  
    id  
    library(limit: $limit, orderBy: ADDED_ON_DESC, page: $page) {  
      results {  
        ...basicBook  
      }  
    }  
  }  
};
```

```
    pageInfo {
      hasNextPage
      page
    }
  }
}

${basicBook}
`;
```

As the final step, we'll import the new query in the Home page component and use it to render the BookGrid component containing the logged-in user's library results:

*client/src/components/Home/index.js*

```
import { useQuery } from "@apollo/client";

import { GetViewerLibrary } from "../../graphql/queries";
import BookGrid from "../../components/BookGrid";
import Button from "../../components/Button";
import Loader from "../../components/Loader";
import PageNotice from "../../components/PageNotice";
import MainLayout from "../../components/MainLayout";

function Home() {
  const { data, error, loading } = useQuery(GetViewerLibrary, {
    variables: { limit: 12, page: 1 }
  });

  let content = null;

  if (loading && !data) {
    content = <Loader centered />;
  } else if (data?.viewer && !data.viewer.library.results.length) {
    content = (
      <div className="flex flex-col h-full justify-center text-center">
        <p className="text-gray-500 text-2xl">
          Time to add some books to your library!
        </p>
      </div>
    );
  } else if (data?.viewer) {
```

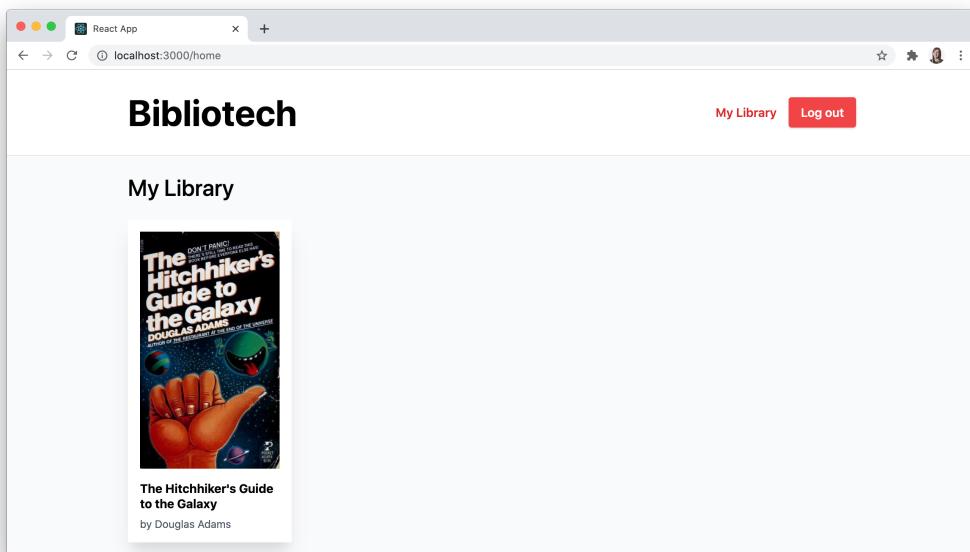
```
const {
  pageInfo: { hasNextPage },
  results
} = data.viewer.library;

content = (
  <>
    <div className="mb-8">
      <h2 className="font-medium mb-6 text-3xl">Your Library</h2>
      <BookGrid books={results} />
    </div>
    {hasNextPage && (
      <div className="flex justify-center">
        <Button text="Load More" type="button" />
      </div>
    )}
  </>
);
} else if (!data.viewer || error) {
  content = <PageNotice text="Book list is unavailable. Please try again." />;
}

return <MainLayout>{content}</MainLayout>;
}

export default Home;
```

After authenticating a user, the `/home` route should render only the books they have added to their library:



## Summary

With Apollo Client configured and client-side authentication in place, we have now reached the end of one of the most ambitious chapters yet. We began by setting up Apollo Client in the React application and using it to fetch a list of books for the index page. We then refactored our server-side authentication code to send an `HttpOnly` cookie to the browser to store a user's JWT (noting that additional security measures should be layered on top of this to protect against XSS and CSRF attacks in production).

We then created an `AuthContext` to manage authentication state and make that state available throughout the application. Finally, we added a few more features to the user interface by allowing users to log out, wrapping React Router's `Route` component to redirect users as needed depending on their authentication state, and adding the user's library books to the homepage.

In the next chapter, we continue building out additional features of the user interface, including pagination for the book lists on the index page and homepage, a route for displaying detailed book information, and the ability to create, update, and delete book reviews.

## Chapter 9

# Pagination, Mutations, and the Apollo Client Cache

In this chapter, we will:

- Customize how data is stored in Apollo Client’s normalized cache
- Paginate lists of books on the index and home pages
- Add a route to display a single book with its details
- Allow users to add and remove books from their library
- Add a book search form and a route to display results
- Allow users to create, update, and delete their book reviews
- Create a form that allows users to submit new books

## Paginate the List of Books on the Index Page

Bibliotech’s React application has started to take shape now, and by the end of this chapter the majority of its features will be in place. Our next step will be to load more results when a user clicks the “Load More” button on the index page. We’ll lay some essential groundwork first and then learn about how the Apollo Client 3 cache works so we can properly merge additional results into this normalized cache and trigger a re-render of the user interface.

As a starting point, we’ll update the `Index` page component so that it stores the `limit` argument value in a variable inside the component. We’ll also destructure the built-in `fetchMore` function from `useQuery` so that we can use it to send follow-up queries for additional pages of results when the “Load More” button is clicked:

`client/src/pages/Index/index.js`

```
// ...
```

```
function Index() {
  const limit = 12;
  const { data, error, fetchMore, loading } = useQuery(GetBooks, {
    variables: { limit, page: 1 }
  });

  // ...
}

export default Index;
```

Now we can update the `Button` component to call the `fetchMore` function in its `onClick` prop:

`client/src/pages/Index/index.js`

```
// ...

function Index() {
  // ...

  let content = null;

  if (loading && !data) {
    content = <Loader centered />;
  } else if (data?.books) {
    const {
      pageInfo: { hasNextPage, page },
      results
    } = data.books;

    content = (
      <>
        <div className="mb-8">
          <BookGrid books={results} />
        </div>
        {hasNextPage && (
          <div className="flex justify-center">
            <Button
              onClick={() => {
                fetchMore({
                  variables: { limit, page: page + 1 }
                });
              }}
            >Load More</Button>
          </div>
        )}
      </>
    );
  }
}

export default Index;
```

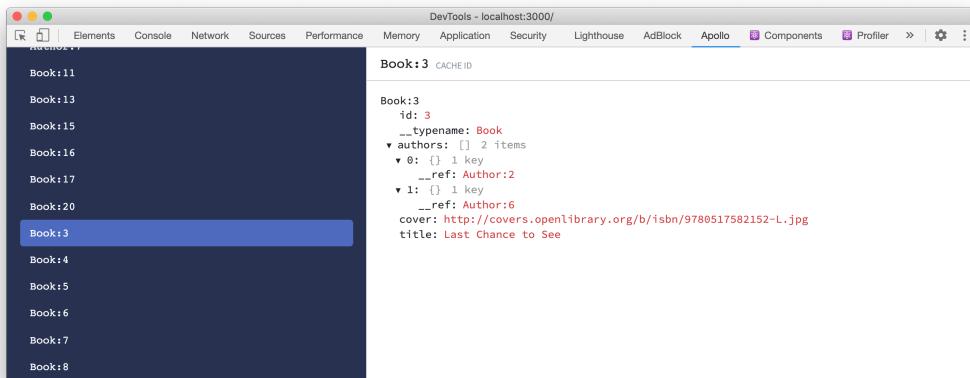
```
        }
      text="Load More"
      type="button"
    />
  </div>
)
);
} else if (error) {
  content = <PageNotice text="Book list is unavailable. Please try again." />;
}

return <MainLayout>{content}</MainLayout>;
}

export default Index;
```

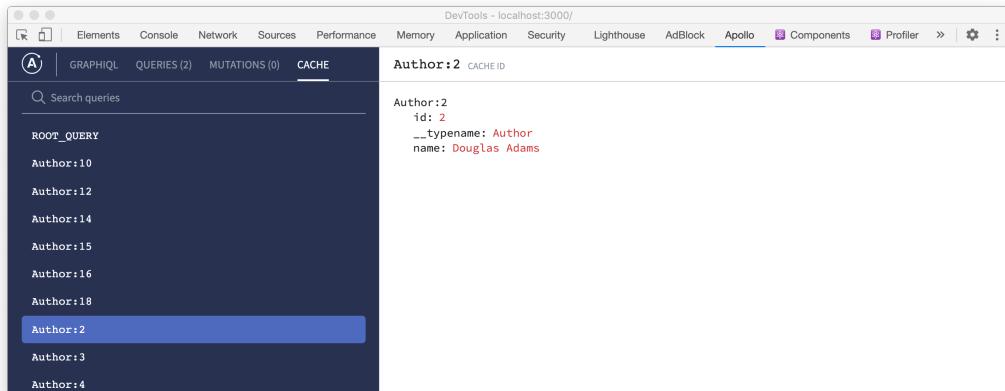
Under the hood, Apollo Client will now resend the `GetBooks` query for the next page of results when the user clicks the button. However, the new books won't be rendered on the page until we provide Apollo Client with some direction about how to merge those results into its normalized cache. We haven't concerned ourselves much with the `InMemoryCache` we passed into Apollo Client in the last chapter, but we'll need to take a closer look at it now.

With the Apollo Dev Tools installed in a browser, we can view the data currently stored in the cache via the cache inspector:



Here we can see the book object represented mostly as we would expect, but it also contains a `__typename` field (which was added by Apollo Client) and the array of authors appears to contain

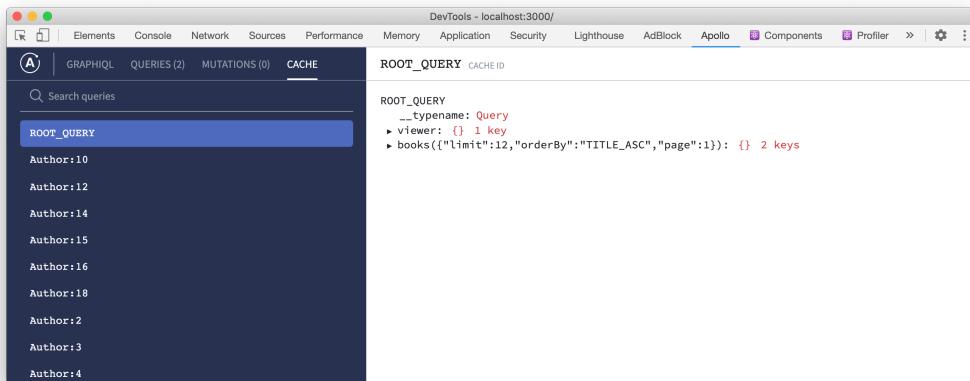
objects with foreign key-like references to the relevant authors instead of the full selection of author fields as we received as data in the query response. Scrolling through the cache, we can eventually find some author objects as well:



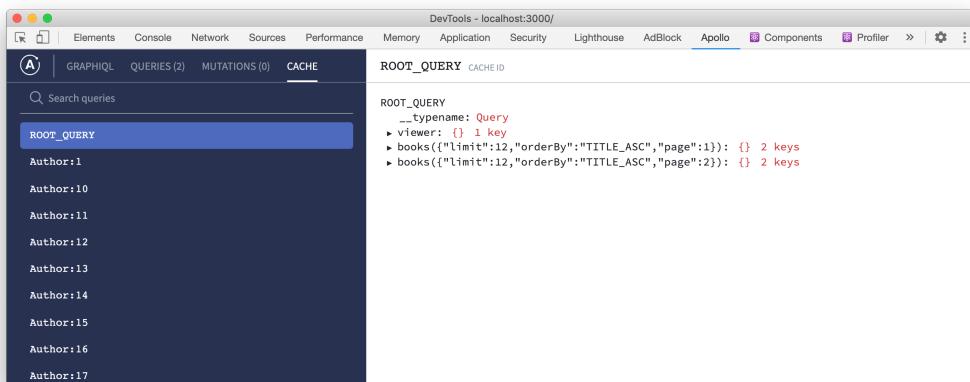
What we have seen here is the “normalized” aspect of Apollo Client’s cache in action. Apollo Client will generate a unique ID for the identifiable objects included in a query response and then store that data in a flat lookup table. By default, and as we can see above, the cache will construct a cache key from the `__typename` and the `id` or `_id` field of an object, if it’s available. Otherwise, we can specify a different unique identifier field for the object in the cache options. This is why we have so far always included the `id` field for Object types in query selection sets even when it isn’t required directly in the component.

Later on, if an updated version of a cached object is received in another query, then it will be merged with the existing object.<sup>1</sup> Caching response data in this manner can help us avoid sending requests to the server for later queries when the data may be retrieved locally from the in-memory cache. Now let’s take a look at what get’s cached for the query root operation type after the first page of results is rendered on the index page:

<sup>1</sup><https://www.apollographql.com/docs/react/caching/cache-configuration/#data-normalization>



If we click the “Load More” button, then the cache will update with another books entry:



We can see now that the two pages of book results end up on two different fields on the `ROOT_QUERY` cache object, rather than being merged like an updated author or book object as we might expect. The reason is that the Apollo Client cache has no way of knowing how to merge fields with different combinations of arguments or complex lists of results. What Apollo Client 3 does have, however, is an API that allows us to explain to the cache how we want these results to be merged and stored as a single field using a *field policy*. To define field policies, we'll create a new file called `typePolicies.js` in `client/src/graphql` and add the following code to it:

`client/src/graphql/typePolicies.js`

```
const typePolicies = {
  Query: {
```

```
    fields: {
      books: {
        // Field policy for the books query will go here...
      }
    }
};

export default typePolicies;
```

Ultimately, we want all pages of book results for this query to be contained within a single field on the ROOT\_QUERY cache object. That means that pagination arguments won't be relevant to how this field should be cached, so we can disable them as follows:

*client/src/graphql/typePolicies.js*

```
const typePolicies = {
  Query: {
    fields: {
      books: {
        keyArgs: false
      }
    }
  }
};

export default typePolicies;
```

We can then pass our new `typePolicies` into the `inMemoryCache`:

*client/src/graphql/apollo.js*

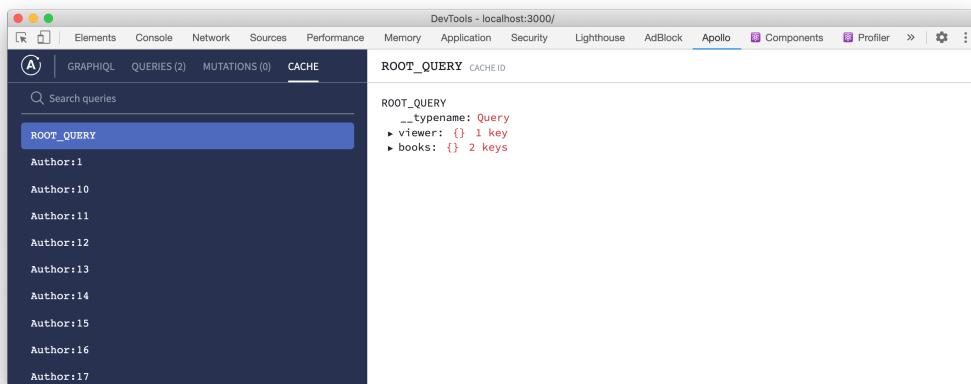
```
import { ApolloClient, HttpLink, InMemoryCache } from "@apollo/client";

import typePolicies from "./typePolicies";

const client = new ApolloClient({
  cache: new InMemoryCache({ typePolicies }),
  // ...
});

export default client;
```

When the application reloads, we'll be able to see that the `book` field is now cached without the arguments and their values:



But alas, we have a new issue! When a user clicks the “Load More” button, the second page of results will now overwrite the first page of results in the user interface, but what we want is for the second page of results to be appended to the bottom of the first page. To fix this issue, we need to define a `merge` function in the field policy as well. This function will explain to Apollo Client how to merge the `results` field of the existing list of books with the incoming page of books, update the `pageInfo` to the incoming page’s values, and set the `__typename` as the cache expects:

`client/src/graphql/apollo.js`

```
const typePolicies = {
  Query: {
    fields: {
      books: {
        keyArgs: false,
        merge(existing, incoming, { args: { page, limit } }) {
          const { __typeName, pageInfo, results } = incoming;
          const mergedResults = existing?.results?.length
            ? existing.results.slice(0)
            : [];

          for (let i = 0; i < results.length; ++i) {
            mergedResults[(page - 1) * limit + i] = results[i];
          }
          return { results: mergedResults, pageInfo, __typeName };
        }
      }
    }
  }
}
```

```
    }
  }
};

export default typePolicies;
```

With this code in place, Apollo Client will now automatically re-render the `BookGrid` component when the cache's `book` field has new results merged into it.

## Display a Single Book with Details

We'll now shift our attention to a completely new user interface feature—a route to display detailed information about a single book. Using URL parameters with React Router, we will format that route as `/book/:id` where `:id` represents the unique ID of a book. Using this URL parameter, we'll be able to pass the ID from the route to a query variable when we fetch the book from the GraphQL API.

Before writing the new query to fetch the book data, we need to do a bit of setup first. The book page will include a paginated list of related reviews sorted in descending order by date. To render the dates in a human-readable format, we'll install `Day.js` in the `client` directory:

```
npm i dayjs@1.10.4
```

The book page will also include `h2` and `h3` elements to render the book title and review section headings respectively, so we'll add some generic styles for these elements to `index.css`:

`client/src/index.css`

```
@tailwind base;

h1 {
  @apply font-bold leading-none text-5xl;
}

h2 {
  @apply font-bold leading-none text-4xl text-gray-700;
}

h3 {
  @apply font-bold leading-none text-3xl text-gray-700;
}
```

```
/* ... */
```

Next, we'll add a new query called `GetBook` to `queries.js`. We'll reuse our `basicBook` fragment once again, but also add the `summary` and paginated `reviews` fields to the selection set:

`client/src/graphql/queries.js`

```
import { gql } from "@apollo/client";

import { basicBook } from "./fragments";

export const GetBook = gql`  
query GetBook($id: ID!, $reviewsLimit: Int, $reviewsPage: Int) {  
  book(id: $id) {  
    ...basicBook  
    summary  
    reviews(  
      limit: $reviewsLimit  
      orderBy: REVIEWED_ON_DESC  
      page: $reviewsPage  
    ) {  
      results {  
        id  
        book {  
          id  
        }  
        reviewedOn  
        rating  
        reviewer {  
          id  
          name  
        }  
        text  
      }  
      pageInfo {  
        hasNextPage  
        page  
      }  
    }  
  }  
}
```

```
 ${basicBook}
`;
// ...
```

As we learned in the previous section, when we fetch additional pages of results, we need to specify a field policy that instructs Apollo Client how to standardize the cache key for the field as well as how to merge those results into the cache. The logic for merging the `reviews` field results will be the same as the `books` query, so let's take this opportunity to refactor that code into a utility function called `mergePageResults`, and then apply the return value of that function as the field policy for each of the fields instead:

*client/src/graphql/typePolicies.js*

```
function mergePageResults(keyArgs = false) {
  return {
    keyArgs,
    merge(existing, incoming, { args: { page, limit } }) {
      const { __typeName, pageInfo, results } = incoming;
      const mergedResults = existing?.results?.length
        ? existing.results.slice(0)
        : [];

      for (let i = 0; i < results.length; ++i) {
        mergedResults[(page - 1) * limit + i] = results[i];
      }
      return { results: mergedResults, pageInfo, __typeName };
    }
  };
}

const typePolicies = {
  Book: {
    fields: {
      reviews: mergePageResults()
    }
  },
  Query: {
    fields: {
      books: mergePageResults()
    }
  }
}
```

```
};

export default typePolicies;
```

Before we create the page component that will correspond to the `/book/:id` route, let's add a `ReviewList` component that will render the paginated list of reviews. Create a `ReviewList` component in `client/src/components` and add an `index.js` file to it with this code:

`client/src/components/ReviewList/index.js`

```
import dayjs from "dayjs";

function ReviewsList({ reviews }) {
  return reviews.map(({ id, rating, reviewedOn, reviewer, text }) => (
    <div className="pt-10" key={id}>
      <div className="sm:flex sm:justify-between mb-4 sm:mb-0">
        <div>
          <p>
            <span className="font-bold">
              {reviewer.name}
            </span>{" "}
            {rating && `rated this book ${rating}/5`}
          </p>
          <p className="text-gray-600 text-sm mb-4">
            Reviewed on {dayjs(reviewedOn).format("MMM D, YYYY")}
          </p>
        </div>
        <div>
          <p>{text}</p>
        </div>
      </div>
    );
}

export default ReviewsList;
```

Now we can create our new Book component in `client/src/pages`. Let's set up the component in the `index.js` file without rendering the `ReviewList` first:

`client/src/pages/Book/index.js`

```
import { useParams } from "react-router-dom";
import { useQuery } from "@apollo/client";
```

```
import { GetBook } from "../../graphql/queries";
import Button from "../../components/Button";
import Loader from "../../components/Loader";
import MainLayout from "../../components/MainLayout";
import PageNotice from "../../components/PageNotice";
import ReviewsList from "../../components/ReviewList";

function Book() {
  const { id } = useParams();
  const reviewsLimit = 20;

  const { data, error, fetchMore, loading } = useQuery(GetBook, {
    variables: { id, reviewsLimit, reviewsPage: 1 }
  });

  let content = null;

  if (loading && !data) {
    content = <Loader centered />;
  } else if (data?.book) {
    const {
      book: { authors, cover, reviews, summary, title }
    } = data;

    content = (
      <div className="bg-white p-8 shadow-md">
        <div className="flex flex-col sm:flex-row items-center
          sm:items-start sm:justify-between border-b border-gray-300
          border-solid pb-8">
          {cover ? (
            <img
              className="mb-4 sm:mb-0 w-40 sm:w-1/4 sm:order-2"
              src={cover}
              alt={`${title} cover`}
            />
          ) : (
            <div className="bg-gray-200 flex flex-none justify-center
              items-center mb-4 sm:mb-0 py-4 w-40 sm:w-1/4 sm:order-2">
              <span className="italic px-8 py-2 md:py-24 lg:py-32
                text-center text-gray-600">
                Cover image unavailable
              </span>
            </div>
          )
        </div>
      </div>
    );
  }
}

export default Book;
```

```

        </span>
      </div>
    )}
<div className="sm:mr-8 sm:order-1 text-center sm:text-left">
  <h2>{title}</h2>
  <p className="leading-tight my-4 text-gray-600 text-lg">{`by
    ${authors
      .map(author => author.name)
      .join(", ")}`}</p>
  {summary ? (
    <p className="mb-4">{summary}</p>
  ) : (
    <p className="mb-4 italic text-gray-400">
      Book summary unavailable.
    </p>
  )}
</div>
</div>
 {/* List of reviews will go here... */}
</div>
);
} else if (error) {
  content = <PageNotice text="Book not found!" />;
}

return <MainLayout>{content}</MainLayout>;
}

export default Book;

```

Above, we use React Router's `useParams` hook to extract the `id` URL parameter from the route. We then pass it to `useQuery` as the value of the `$id` variable. Note that we again destructure the `fetchMore` function from `useQuery` so we can fetch additional pages of reviews. Now let's add the `ReviewList` to the component, along with a "Load More" button to get more reviews:

*client/src/pages/Book/index.js*

```

// ...

function Book({ match }) {
  const { id } = useParams();
  const reviewsLimit = 20;

```

```
const { data, error, fetchMore, loading } = useQuery(GetBook, {
  variables: { id, reviewsLimit, reviewsPage: 1 }
});

let content = null;

if (loading && !data) {
  content = <Loader centered />;
} else if (data?.book) {
  const {
    book: { authors, cover, reviews, summary, title }
  } = data;

  content = (
    <div className="bg-white p-8 shadow-xl">
      {/* ... */}
      <div className="mt-8">
        <h3 className="mb-4 sm:mb-0">What Readers Say</h3>
        {reviews.results?.length ? (
          <div>
            <ReviewsList reviews={reviews.results} />
            {reviews.pageInfo.hasNextPage && (
              <Button
                className="mt-4"
                onClick={() => {
                  fetchMore({
                    variables: {
                      reviewsLimit,
                      reviewsPage: reviews.pageInfo.page + 1
                    }
                  });
                }}
              >Load More</Button>
            )}
          </div>
        ) : (
          <p className="italic mt-4">No reviews for this book yet!</p>
        )}
      </div>
    
```

```
        </div>
    );
} else if (error) {
    content = <PageNotice text="Book not found!" />;
}

return <MainLayout>{content}</MainLayout>;
}

export default Book;
```

Our Book page component is ready to go now so let's add it as a route:

*client/src/router/index.js*

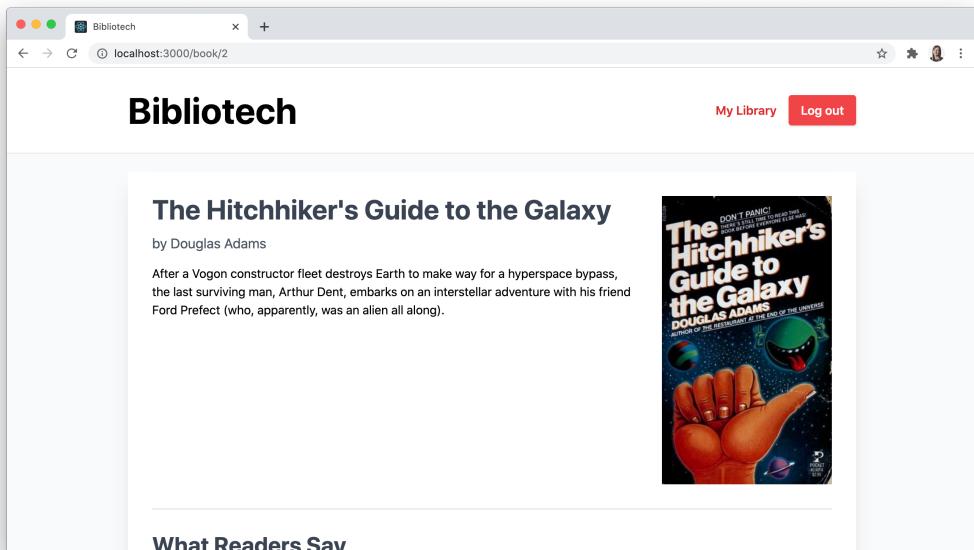
```
import { Switch } from "react-router";

import Book from "../pages/Book";
// ...

export function Routes() {
    return (
        <Switch>
            {/* ... */}
            <PublicRoute exact path="/book/:id" component={Book} />
        </Switch>
    );
}

export default Routes;
```

Try navigating to a page for an existing book. It should render like this one does:



To make it easier for users to navigate to book pages without knowing what the book IDs are, let's also update the `BookGrid` component so that it will render the appropriate book page for users whenever they click on a book item in the list:

`client/src/components/BookGrid/index.js`

```
import { useHistory } from "react-router-dom";

function BookGrid({ books }) {
  const history = useHistory();

  return (
    <ul className="gap-8 grid grid-cols-1 sm:grid-cols-2 md:grid-cols-3
      lg:grid-cols-4">
      {books.map(({ authors, cover, id, title }) => (
        <li
          className="bg-white cursor-pointer flex flex-col justify-end p-4
            shadow-xl"
          key={id}
          onClick={() => {
            history.push(`/book/${id}`);
          }}
        >
```

```
        {/* ... */}
      </li>
    ))
</ul>
);
}

export default BookGrid;
```

With this code in place, we should be able to navigate to any book detail page after clicking on an item in the BookGrid.

## Add and Remove Books from a Library

The new book page is shaping up, but it's still missing an important feature. When an authenticated user views the details for an individual book, they need to be able to add or remove it from their library as well. That means we finally get to add some new mutations to the React application. We'll start by adding `AddBooksToLibrary` and `RemoveBooksFromLibrary` mutations to `mutations.js`:

`client/src/graphql/mutations.js`

```
// ...

export const AddBooksToLibrary = gql` 
  mutation AddBooksToLibrary($input: UpdateLibraryBooksInput!) {
    addBooksToLibrary(input: $input) {
      id
    }
  }
`;

// ...

export const RemoveBooksFromLibrary = gql` 
  mutation RemoveBooksFromLibrary($input: UpdateLibraryBooksInput!) {
    removeBooksFromLibrary(input: $input) {
      id
    }
  }
`;
```

```
// ...
```

Before we put these mutations to use, we need to pause and consider the different states in which a book page may be rendered. The `/book/:id` route is public, so the page may be rendered for an unauthenticated user and we wouldn't want to display any kind of user interface element that would suggest they can add a book to a library. If a user is authenticated but they don't have the book in their library, then we'll show them a button that allows them to add the book and trigger the `AddBooksToLibrary` mutation. Alternatively, if an authenticated user has the book in their library already, then we'll show them a button that allows them to remove the book and that sends the `RemoveBooksFromLibrary` mutation instead.

We can handle conditionally rendering a button for authenticated users using the existing `AuthContext`, but the matter of calculating whether the viewer has the book in their library is a bit more complicated. We could leave it up to the client to try to sort this out, but this seems like a great opportunity to shift this work to the server and add a field to the `Book` Object type in the GraphQL API so that clients can simply request this field and render the button as needed.

To do this, we'll add a boolean `viewerHasInLibrary` field to the `Book` type definition:

`server/src/graphql/typeDefs.js`

```
import { gql } from "apollo-server-express";

const typeDefs = gql`  
# ...  
  
"""  
A written work that can be attributed to one or more authors and can be  
reviewed by users.  
"""  
type Book {  
# ...  
  "A boolean indicating if the viewing user has added this book to their  
library."  
  viewerHasInLibrary: Boolean  
}  
  
# ...  
`;  
  
export default typeDefs;
```

To resolve this field, we'll have to query the REST API's `/userBooks` endpoint to determine if a record already exists that contains the user's ID and the book ID. Let's add a `checkViewerHasInLibrary` method to the `JsonServerApi` data source to assist with that:

*server/src/graphql/datasources/JsonServerApi.js*

```
// ...

class JsonServerApi extends RESTDataSource {
    // ...

    async checkViewerHasInLibrary(id, bookId) {
        const response = await
            this.get(`/userBooks?userId=${id}&bookId=${bookId}`);
        return !!response.length;
    }

    // ...
}

export default JsonServerApi;
```

Then we'll update our resolvers:

*server/src/graphql/resolvers.js*

```
// ...

const resolvers = {
    // ...
    Book: {
        // ...
        reviews(book, args, { dataSources }, info) {
            return dataSources.jsonServerApi.getBookReviews(book.id, args);
        },
        viewerHasInLibrary(book, args, { dataSources, user }, info) {
            return user?.sub
                ? dataSources.jsonServerApi.checkViewerHasInLibrary(user.sub,
                    book.id)
                : null;
        }
    },
    // ...
```

```
};

export default resolvers;
```

Back on the client side, we'll add the new `viewerHasInLibrary` field to the `GetBook` query:

`client/src/graphql/queries.js`

```
// ...

export const GetBook = gql`query GetBook($id: ID!, $reviewsLimit: Int, $reviewsPage: Int) {  book(id: $id) {    ...basicBook    summary    viewerHasInLibrary    # ...
  }
${basicBook}
`;
```

Now we'll make our first pass at updating the Book page component to conditionally render a button below the book summary for authenticated users only. This button will allow those users to add or remove a book depending on whether they already have the book in their library:

`client/src/pages/Book/index.js`

```
// ...

import { GetBook } from "../../graphql/queries";
import { useAuth } from "../../context/AuthContext";
// ...

function Book() {
  const { id } = useParams();
  const { viewer } = useAuth();
  const reviewsLimit = 20;

// ...
```

```
if (loading && !data) {
  content = <Loader centered />;
} else if (data?.book) {
  const {
    book: { authors, cover, reviews, summary, title, viewerHasInLibrary }
  } = data;

  content = (
    <div className="bg-white p-8 shadow-xl">
      <div className="flex flex-col sm:flex-row items-center
        sm:items-start sm:justify-between border-b border-gray-300
        border-solid pb-8">
        {/* ... */}
        <div className="sm:mr-8 sm:order-1 text-center sm:text-left">
          <h2>{title}</h2>
          <p className="leading-tight my-4 text-gray-600 text-lg">{`by
            ${authors}
            .map(author => author.name)
            .join(", ")}`}</p>
        {/* ... */}
        {viewer && (
          <Button
            className="mt-4"
            onClick={() => {
              // Run the mutation to add or remove the book here...
            }}
            text={
              viewerHasInLibrary ? "Remove from Library" : "Add to
                Library"
            }
          />
        )}
      </div>
    </div>
    {/* ... */}
  </div>
);
} else if (error) {
  content = <PageNotice text="Book not found!" />;
}
```

```
    return <MainLayout>{content}</MainLayout>;
}

export default Book;
```

As the next step, we'll add the `AddBooksToLibrary` and `RemoveBooksFromLibrary` mutations to the `Book` page component and conditionally send each mutation on the button click:

*client/src/pages/Book/index.js*

```
import { useParams } from "react-router-dom";
import { useMutation, useQuery } from "@apollo/client";

import {
  AddBooksToLibrary,
  RemoveBooksFromLibrary
} from "../../graphql/mutations";
// ...

function Book() {
  // ...
  const [addBooksToLibrary] = useMutation(AddBooksToLibrary);
  const [removeBooksFromLibrary] = useMutation(RemoveBooksFromLibrary);

  let content = null;

  if (loading && !data) {
    content = <Loader centered />;
  } else if (data?.book) {
    const {
      book: { authors, cover, reviews, summary, title, viewerHasInLibrary }
    } = data;

    content = (
      <div className="bg-white p-8 shadow-xl">
        <div className="flex flex-col sm:flex-row items-center sm:items-start sm:justify-between border-b border-gray-300 border-solid pb-8">
          {/* ... */}
          <div className="sm:mr-8 sm:order-1 text-center sm:text-left">
            {/* ... */}
            {viewer && (
              <button onClick={handleAddToLibrary} type="button">
                Add to Library
              </button>
            )}
          </div>
        </div>
      </div>
    );
  }
}

export default Book;
```

```
        <Button
            className="mt-4"
            onClick={() => {
                const variables = {
                    input: { bookIds: [id], userId: viewer.id }
                };

                if (viewerHasInLibrary) {
                    removeBooksFromLibrary({ variables });
                } else {
                    addBooksToLibrary({ variables });
                }
            }}
            text={
                viewerHasInLibrary ? "Remove from Library" : "Add to
                Library"
            }
        />
    )}
</div>
</div>
 {/* ... */}
</div>
);
} else if (error) {
    content = <PageNotice text="Book not found!" />;
}

return <MainLayout>{content}</MainLayout>;
}

export default Book;
```

We run the mutation now by clicking the button and we'll be able to see that records are added and removed from the db.json file. However, we have just introduced a bug into the React application because the value of `viewerHasInLibrary` isn't automatically updated after the mutation. As a result, what's rendered on the screen becomes out of sync with reality and a user will send the same mutation if the button is clicked more than once (that is, until the page is fully reloaded and the book details are refetched).

To fix this bug, we'll need to pass an `update` option into the `useMutation` hook for each of the mutations. The `update` option allows us to update the Apollo Client cache after the mutation successfully runs so the user interface will be re-rendered based on the correct state.

Because both mutations will need to update the cache in the same way, we'll create a reusable `updateViewerHasInLibrary` function that takes the cache and the ID of the book as parameters. This function will use the `cache` object's `readQuery` and `writeQuery` methods to get the appropriate book object from the cache and then update its `viewerHasInLibrary` field. To create this function, we'll make a new directory called `utils` in `client/src` and add a file called `updateQueries.js` to it with the following code:

`client/src/utils/updateQueries.js`

```
import { GetBook } from "../graphql/queries";

export function updateViewerHasInLibrary(cache, id) {
  const { book } = cache.readQuery({
    query: GetBook,
    variables: { id }
  });

  cache.writeQuery({
    query: GetBook,
    data: {
      book: {
        ...book,
        viewerHasInLibrary: !book.viewerHasInLibrary
      }
    }
  });
}
```

We can then import the function into the `Book` page component and call it inside the `update` method option for each mutation:

`client/src/pages/Book/index.js`

```
// ...
import { updateViewerHasInLibrary } from "../../utils/updateQueries";
// ...

function Book() {
  // ...
  const [addBooksToLibrary] = useMutation(AddBooksToLibrary, {
    update: cache => {
      updateViewerHasInLibrary(cache, id);
    }
}
```

```
});  
const [removeBooksFromLibrary] = useMutation(RemoveBooksFromLibrary, {  
  update: cache => {  
    updateViewerHasInLibrary(cache, id);  
  }  
});  
  
// ...  
}  
  
export default Book;
```

Now the text on the library-updating button will toggle back and forth after a mutation completes, and the correct mutation will subsequently be sent if the button is clicked more than once.

## Add Pagination to the Home Page

Now that users have the power to add and remove books from their libraries, these lists of books may grow quite long on the homepage, so let's quickly add pagination for the books query on the /home route as well. To do this, we'll update the Home page component much as we did when we added pagination to the index page:

*client/src/pages/Home/index.js*

```
// ...  
  
function Home() {  
  const limit = 12;  
  const { data, error, fetchMore, loading } = useQuery(GetViewerLibrary, {  
    variables: { limit, page: 1 }  
});  
  
  let content = null;  
  
  if (loading && !data) {  
    content = <Loader centered />;  
  } else if (data?.viewer && !data.viewer.library.results.length) {  
    // ...  
  } else if (data?.viewer) {  
    const {  
      pageInfo: { hasNextPage, page },  
      results:  
    } = data.viewer.library;  
    content =   
      <List items={results} key={page}>  
        {results.map(book =>   
          <Book book={book} key={book.id} />)  
        }  
      </List>  
  }  
}  
  
export default Home;
```

```
    results
} = data.viewer.library;

content = (
  <>
  {/* ... */}
  {hasNextPage && (
    <div className="flex justify-center">
      <Button
        text="Load More"
        onClick={() => {
          fetchMore({
            variables: { limit, page: page + 1 }
          });
        }}
        type="button"
      />
    </div>
  )}
  </>
);
} else if (!data.viewer || error) {
  content = <PageNotice text="Book list is unavailable. Please try again." />;
}

return <MainLayout>{content}</MainLayout>;
}

export default Home;
```

And to update the Apollo Client cache after fetching additional library books, we'll add another field policy to `typePolicies.js`. However, we'll need to take a different approach with this policy than we did with the previous two. Rather than applying it directly to the `viewer` field, we'll add it to the `library` field on the applicable `User` in the cache because the `viewer` field on the `ROOT_QUERY` object will only contain a reference to the normalized `User` object:

`client/src/graphql/typePolicies.js`

```
// ...

const typePolicies = {
```

```
Book: {
  fields: {
    reviews: mergePageResults()
  }
},
User: {
  fields: {
    library: mergePageResults()
  }
},
Query: {
  fields: {
    books: mergePageResults()
  }
}
};

export default typePolicies;
```

If the currently authenticated user has fewer than 12 books in their library, try adjusting the `limit` constant in the `Home` page component temporarily to confirm that the pagination works.

## Add Book Search

Before we turn our attention back to mutations for the remainder of the chapter, we have one more query-related feature to add. Users likely won't want to sift through pages and pages of alphabetically ordered results to find a specific book, so we still need to add a search form and search results page to make books in the Bibliotech catalog more discoverable.

To begin, we'll add a new `SearchBooks` query to `queries.js` that will fetch books with titles that match a submitted search string, as well as the books authored by people whose names match that same string:

`client/src/graphql/queries.js`

```
// ...

export const SearchBooks = gql`  
query SearchBooks($query: String!) {  
  searchBooks(query: $query, orderBy: RESULT_ASC) {  
    ... on Book {  
      ...basicBook
```

```
        }
      ... on Author {
        books {
          ...basicBook
        }
      }
    }
  ${basicBook}
`;
```

As we did in Chapter 4, we use inline fragments to include both books and authors with their list of books in the response. To build out the user interface elements that will support search, we'll need both a search form to add to the index page and another page component to display search results. We'll also include the same search form at the top of the search results page in case the user wants to do another search.

When the search form is submitted, the string in the search input will be used as a query parameter on the new /search route, and the value of that query parameter will be used as the `query` variable value in the request to the GraphQL API. React Router doesn't have any built-in way to parse query strings, so we'll install another package on `client` to help with this:

```
npm i query-string@7.0.0
```

Now we'll add a new `SearchBooksForm` component that contains a form with a single text-based input and a button. When the form is submitted, the value of the input will be used as the `q` query parameter. And when this form is rendered at the top of the search results page we'll parse the same query parameter to pre-populate the same text in the input for the user's convenience:

`client/src/components/SearchBooksForm/index.js`

```
import { useHistory, useLocation } from "react-router-dom";
import { useState } from "react";
import queryString from "query-string";

import Button from "../Button";
import TextInput from "../TextInput";

function SearchBooksForm() {
  const history = useHistory();
  const location = useLocation();

  const { q } = queryString.parse(location.search);
```

```
const [query, setQuery] = useState(q || "");

return (
  <div className="flex justify-center mb-4 w-full">
    <form
      className="flex flex-col sm:flex-row items-center justify-center
      max-w-xl w-full"
      onSubmit={event => {
        event.preventDefault();
        history.push(`/search?q=${query}`);
      }}
    >
      <TextInput
        className="flex-auto sm:max-w-xs mb-4 sm:mb-0 sm:mr-2 w-10/12"
        hiddenLabel
        id="query"
        inputWidthClass="w-full"
        label="Search for a book title or author"
        name="query"
        onChange={event => {
          setQuery(event.target.value);
        }}
        placeholder="Search for a book title or author"
        type="search"
        value={query}
      />
      <Button primary text="Search" type="submit" />
    </form>
  </div>
);

export default SearchBooksForm;
```

Now we can set up the search results page. Create a `Search` directory in `client/src/pages` and add an `index.js` file to it with the following code:

`client/src/pages/Search/index.js`

```
import { useLocation } from "react-router-dom";
import { useQuery } from "@apollo/client";
import queryString from "query-string";
```

```
import { SearchBooks } from "../../graphql/queries";
import BookGrid from "../../components/BookGrid";
import Loader from "../../components/Loader";
import MainLayout from "../../components/MainLayout";
import PageNotice from "../../components/PageNotice";
import SearchBooksForm from "../../components/SearchBooksForm";

function Search() {
  const { location } = useLocation();
  const { q } = queryString.parse(location.search);

  const { data, error, loading } = useQuery(SearchBooks, {
    variables: { query: q }
  });

  // Render search results here...
}

export default Search;
```

When this component loads, we parse the `q` query parameter at the end of the route's URL and use its value to send the `SearchBooks` query to the GraphQL API. When matching results are found, the books with matching titles will be directly available in the `data.searchBooks` array, but the books returned for authors with matching names will be nested under a `books` key in the author result, so we need to make a quick pass over the array to flatten all of the book results:

`client/src/pages/Search/index.js`

```
// ...

function Search() {
  // ...

  let content = null;

  if (loading) {
    content = <Loader centered />;
  } else if (data?.searchBooks) {
    const parsedBooks = data.searchBooks.reduce((acc, curr) => {
      if (curr.__typename === "Author") {
        return acc.concat(curr.books);
      }
    }, []);
    content = (
      <BookGrid books={parsedBooks} />
      <PageNotice message="Search results" />
    );
  }
}

export default Search;
```

```
        }
        return acc.push(curr);
    }, []);
}

content = (
    <div className="mb-8">
        <SearchBooksForm />
        {parsedBooks.length ? (
            <BookGrid books={parsedBooks} />
        ) : (
            <p className="mt-8 text-center">
                No books found! Please search again.
            </p>
        )}
    </div>
);
} else if (error) {
    content = (
        <PageNotice text="Book search is unavailable. Please try again." />
    );
}

return <MainLayout>{content}</MainLayout>;
}

export default Search;
```

The Search page component is ready to go, so let's set it up as a publicly available route:

*client/src/router/index.js*

```
// ...
import Search from "../pages/Search";

export function Routes() {
    return (
        <Switch>
            {/* ... */}
            <PublicRoute exact path="/search" component={Search} />
        </Switch>
    );
}
```

```
export default Routes;
```

Lastly, we can add the `SearchBooksForm` to the index page:

`client/src/pages/Index/index.js`

```
// ...
import SearchBooksForm from "../../components/SearchBooksForm";

function Index() {
    // ...

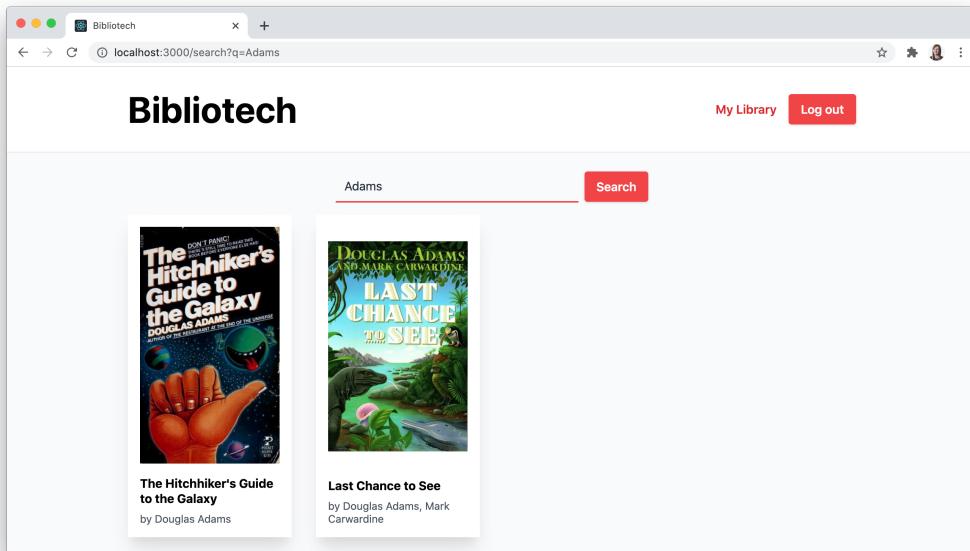
    if (loading && !data) {
        content = <Loader centered />;
    } else if (data?.books) {
        // ...

        content = (
            <>
                <div className="mb-8">
                    <SearchBooksForm />
                    <BookGrid books={results} />
                </div>
                {/* ... */}
            </>
        );
    } else if (error) {
        content = <PageNotice text="Book list is unavailable. Please try again." />;
    }

    return <MainLayout>{content}</MainLayout>;
}

export default Index;
```

Try running a few searches for various book title words and author names. The new search results page should render as follows:



## Create a Book Review

Shifting our focus back to mutations, for our next step we will add a form that submits a mutation to create a new review. To facilitate caching data for new reviews, the fields that we return from the `createReview` mutation will be the same as the fields for the `review results` field in the `book` query, so let's make a reusable fragment to use for both of them now:

`client/src/graphql/fragments.js`

```
// ...  
  
export const fullReview = gql`  
  fragment fullReview on Review {  
    id  
    book {  
      id  
    }  
    reviewedOn  
    rating  
    reviewer {  
      id  
    }  
  }  
`
```

```
    name
  }
  text
}
`;
// ...
```

Now we can set up a new `CreateReview` mutation using the `fullReview` fragment:

`client/src/graphql/mutations.js`

```
import { fullReview, viewerAndToken } from "./fragments";
// ...

export const CreateReview = gql`  
  mutation CreateReview($input: CreateReviewInput!) {  
    createReview(input: $input) {  
      ...fullReview  
    }  
  }  
${fullReview}  
`;  
// ...
```

And we can tidy up the `GetBook` query by importing the same fragment into `queries.js` and using it there as well:

`client/src/graphql/queries.js`

```
import { gql } from "@apollo/client";

import { basicBook, fullReview } from "./fragments";

export const GetBook = gql`  
  query GetBook($id: ID!, $reviewsLimit: Int, $reviewsPage: Int) {  
    book(id: $id) {  
      ...basicBook  
      summary  
      viewerHasInLibrary
```

```
viewerHasReviewed
reviews(
  limit: $reviewsLimit
  orderBy: REVIEWED_ON_DESC
  page: $reviewsPage
) {
  results {
    ...fullReview
  }
  pageInfo {
    hasNextPage
    page
  }
}
${basicBook}
${fullReview}
`;
// ...
```

Users will be able to access the form for book review submissions from the book page. And just as we conditionally rendered some of the book page's components based on whether the user was authenticated and if they already had the book in their library, we will also conditionally render the button that will direct an authenticated user to the book review form, but only if they haven't reviewed the book yet. To assist with this conditional rendering, we'll add a boolean `viewerHasReviewed` field to the `Book` type definition:

`server/src/graphql/typeDefs.js`

```
import { gql } from "apollo-server-express";

const typeDefs = gql` 
  # ...

  """
  A written work that can be attributed to one or more authors and can be
  reviewed by users.
  """
  type Book {
    # ...
```

```
    "A boolean indicating if the viewing user has reviewed the book."
  viewerHasReviewed: Boolean
}

# ...
`;

export default typeDefs;
```

Then we'll add a `checkViewerHasReviewed` method to the `JsonServerApi` data source, similar to the `checkViewerHasInLibrary` method that we previously added:

`server/src/graphql/datasources/JsonServerApi.js`

```
// ...

class JsonServerApi extends RESTDataSource {
  // ...

  async checkViewerHasReviewed(id, bookId) {
    const response = await
      this.get(`/reviews?bookId=${bookId}&userId=${id}`);
    return !!response.length;
  }

  // ...
}

export default JsonServerApi;
```

Now we'll add the resolver for the new `viewerHasReviewed` field:

`server/src/graphql/resolvers.js`

```
// ...

const resolvers = {
  // ...
  Book: {
    // ...
    viewerHasInLibrary(book, args, { dataSources, user }, info) {
      // ...
    }
  }
}

export default resolvers;
```

```
    },
    viewerHasReviewed(book, args, { dataSources, user }, info) {
      return user?.sub
        ? dataSources.jsonServerApi.checkViewerHasReviewed(user.sub,
          book.id)
        : null;
    }
  },
  // ...
};

export default resolvers;
```

And one more time, we'll update the `GetBook` query so that it includes the new field in its selection set now:

`client/src/graphql/queries.js`

```
export const GetBook = gql`query GetBook($id: ID!, $reviewsLimit: Int, $reviewsPage: Int) {
  book(id: $id) {
    ...basicBook
    summary
    viewerHasInLibrary
    viewerHasReviewed
    # ...
  }
}
${basicBook}
`;
```

Moving on, the book review form that we will build contains two fields—a text input for the review's optional text-based content and a select menu for choosing the required numerical book rating. We can reuse the existing `TextInput` component for the text content, but we'll need to add a new `Select` component for the rating field. Create a `Select` directory in `client/src/components` and add an `index.js` file to it with the following code:

`client/src/components>Select/index.js`

```
function Select({
  className,
  hiddenLabel,
```

```
id,
label,
name,
onChange,
options,
selectWidthClass,
value,
...rest
}) {
  return (
    <div className={className}>
      <label
        htmlFor={id}
        className={`block font-semibold ${hiddenLabel && "sr-only"}`}
      >
        {label}
      </label>
      <div className={`${`inline-block relative w-64 ${selectWidthClass}`}`}>
        <select
          className={`block appearance-none w-full bg-white
            focus:bg-gray-100 border-b-2 border-red-500
            hover:border-red-700 px-4 py-2 pr-8 leading-tight
            focus:outline-none w-full`}
          id={id}
          name={name}
          onChange={onChange}
          value={value}
          {...rest}
        >
          {options.map(({ text, value }) => (
            <option key={value} value={value}>
              {text}
            </option>
          )))
        </select>
        <div className="pointer-events-none absolute inset-y-0 right-0 flex
          items-center px-2 text-gray-700">
          <svg
            className="fill-current h-4 w-4"
            xmlns="http://www.w3.org/2000/svg"
            viewBox="0 0 20 20"
          >
```

```
        <path d="M9.293 12.95l.707.707L15.657 8l-1.414-1.414L10 10.828
              5.757 6.586 4.343 8z" />
      </svg>
    </div>
  </div>
);
}

export default Select;
```

Now we're ready to set up a new `ReviewBook` page component containing the book review form. The page that contains this form will be available at the `/book/:bookId/review/new` route, so we can get the `bookId` URL parameter using React Router's `useParams` hook and then use it as the `$bookId` variable value in the mutation later on. We'll also need its `useHistory` hook so we can push the viewer back to the book page after they've submitted the form so they can see their new review. And because we're building another form with controlled components, we'll need to manage some state for each of the form fields:

`client/src/pages/ReviewBook/index.js`

```
import { useHistory, useParams } from "react-router-dom";
import { useMutation } from "@apollo/client";
import { useState } from "react";

import { CreateReview } from "../../graphql/mutations";
import { useAuth } from "../../context/AuthContext";
import Button from "../../components/Button";
import MainLayout from "../../components/MainLayout";
import PageNotice from "../../components/PageNotice";
import Select from "../../components/Select";
import TextInput from "../../components/TextInput";

function ReviewBook() {
  const { bookId } = useParams();
  const { viewer, error: viewerError } = useAuth();
  const history = useHistory();

  const [rating, setRating] = useState("5");
  const [text, setText] = useState("");

  const [createReview] = useMutation(CreateReview, {
```

```
    onCompleted: () => {
      history.push(`/book/${bookId}`);
    }
  });

const handleSubmit = event => {
  event.preventDefault();

  createReview({
    variables: {
      input: {
        bookId,
        rating: parseInt(rating),
        reviewerId: viewer.id,
        text
      }
    }
  }).catch(err => {
    console.error(err);
  });
};

// ...
}

export default ReviewBook;
```

The review form itself will contain the following code:

*client/src/pages/ReviewBook/index.js*

```
// ...

function ReviewBook() {
// ...

let content = null;

if (viewer) {
  content = (
    <div className="bg-white p-8 shadow-xl">
      <h2 className="mb-8">Create a New Review</h2>
```

```
<form onSubmit={handleSubmit}>
  <Select
    className="mb-6"
    id="rating"
    label="Rating"
    name="rating"
    onChange={event => {
      setRating(event.target.value);
    }}
    options={[
      { text: "1", value: "1" },
      { text: "2", value: "2" },
      { text: "3", value: "3" },
      { text: "4", value: "4" },
      { text: "5", value: "5" }
    ]}
    value={rating}
  />
  <TextInput
    className="mb-6 w-full"
    id="text"
    inputWidthClass="w-full"
    label="Your Review"
    name="text"
    onChange={event => {
      setText(event.target.value);
    }}
    placeholder="Write a brief review of the book"
    value={text}
  />
  <Button className="mt-4" primary text="Submit" type="submit" />
</form>
</div>
);
}

} else if (viewerError) {
  content = <PageNotice text="Something went wrong. Please try again!" />;
}

return <MainLayout>{content}</MainLayout>;
}
```

```
export default ReviewBook;
```

Next, we'll wire up a new route for the `ReviewBook` page component:

`client/src/router/index.js`

```
// ...
import ReviewBook from "../pages/ReviewBook";
import Search from "../pages/Search";

export function Routes() {
  return (
    <Switch>
      {/* ... */}
      <PrivateRoute
        exact
        path="/book/:bookId/review/new"
        component={ReviewBook}
      />
      <PublicRoute exact path="/search" component={Search} />
    </Switch>
  );
}

export default Routes;
```

Circling back on the new `viewerHasReviewed` field, we'll update the `Book` page component so that it destructures the new field after loading the book data and then uses the value to determine if an "Add a Review" button should be displayed next to the review section heading:

`client/src/pages/Book/index.js`

```
import { useHistory, useParams } from "react-router-dom";
import { useMutation, useQuery } from "@apollo/client";

// ...

function Book() {
  const { id } = useParams();
  const { viewer } = useAuth();
  const history = useHistory();
```

```
const reviewsLimit = 20;

// ...

let content = null;

if (loading && !data) {
  content = <Loader centered />;
} else if (data?.book) {
  const {
    book: {
      authors,
      cover,
      reviews,
      summary,
      title,
      viewerHasInLibrary,
      viewerHasReviewed
    }
  } = data;

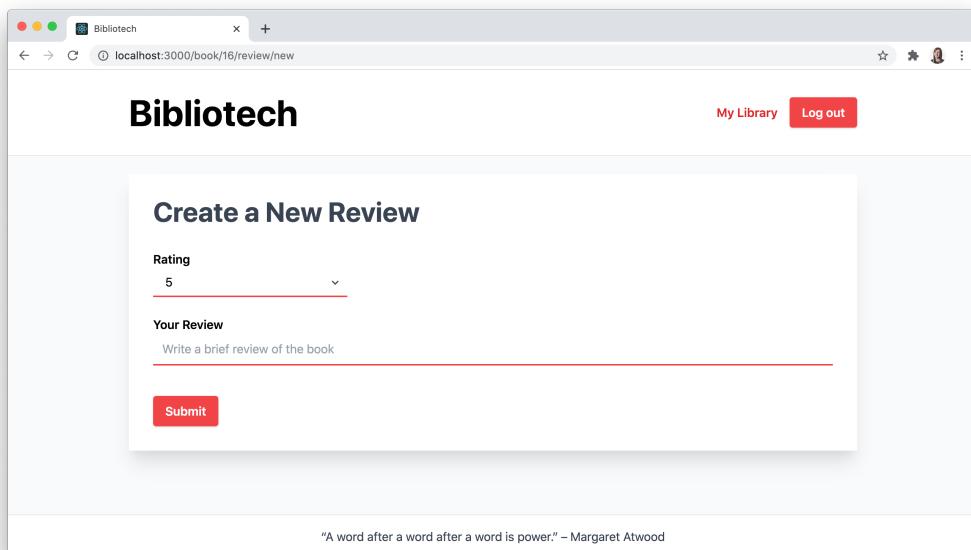
  content = (
    <div className="bg-white p-8 shadow-xl">
      {/* ... */}
      <div className="mt-8">
        <div className="sm:flex sm:justify-between">
          <h3 className="mb-4 sm:mb-0">What Readers Say</h3>
          {viewer && !viewerHasReviewed && (
            <Button
              onClick={() => {
                history.push(`/book/${id}/review/new`);
              }}
              primary
              text="Add a Review"
            />
          )}
        </div>
      {/* ... */}
    </div>
  );
} else if (error) {
```

```
    content = <PageNotice text="Book not found!" />;
}

return <MainLayout>{content}</MainLayout>;
}

export default Book;
```

Try navigating to a book that the authenticated user hasn't reviewed yet and click the "Add a Review" button. The book review form will render as pictured:



The book review form is fully operational, but if we try submitting a review through it now, then we'll encounter what appears to be a bug in the book page's user interface. Specifically, the new review won't appear in the list until we do a full page refresh. So to fix this bug, we'll need to make one more update to the Book page component by setting the `fetchPolicy` option to `cache-and-network` for the `GetBook` query, as well as a `nextFetchPolicy` of `cache-first` when fetching subsequent pages of review results. By choosing these fetch policies, the book page will no longer load with stale data from the cache after the review is submitted:

client/src/pages/Book/index.js

```
// ...
```

```
function Book() {
  // ...

  const { data, error, fetchMore, loading } = useQuery(GetBook, {
    variables: { id, reviewsLimit, reviewsPage: 1 },
    fetchPolicy: "cache-and-network",
    nextFetchPolicy: "cache-first"
  });
  // ...
}

export default Book;
```

By default, Apollo Client uses a `cache-first` fetch policy for all queries to help minimize network requests, but in some instances, we need to override this policy to ensure that the data displayed on the screen and stored in the cache is in sync with the data on the server. You can read more about different fetch policies in the [Apollo Client documentation](#).

With this code in place, the book review form should be fully operational for authenticated users. Try adding at least one new review before moving on to the next section, as it will build on top of this form to support updates to existing reviews too.

## Update a Book Review

Now that the book review form can add reviews, we can reuse the same `ReviewBook` page component to allow users to update their existing reviews as well. Let's begin by adding an `UpdateReview` mutation to `mutations.js`:

`client/src/graphql/mutations.js`

```
// ...

export const UpdateReview = gql`  
mutation UpdateReview($input: UpdateReviewInput!) {  
  updateReview(input: $input) {  
    ...fullReview  
  }  
  ${fullReview}  
};`
```

Next, we'll declare a new route to render the `ReviewBook` page component when it's used to update a book. The path will be very similar to the other route we added for creating new reviews, except this time it will use a `:reviewId` URL parameter at the end of the route instead of `new`:

`client/src/router/index.js`

```
// ...

export function Routes() {
  return (
    <Switch>
      {/* ... */}
      <PrivateRoute
        exact
        path="/book/:bookId/review/:reviewId"
        component={ReviewBook}
      />
      <PublicRoute exact path="/search" component={Search} />
    </Switch>
  );
}

export default Routes;
```

To help users access the form to update their existing reviews, we'll conditionally render an "Update" button in the `ReviewList` component next to any review that they wrote. To do that, we must add a `viewerId` prop to the component so we can compare it to the `id` of the user that wrote each review in the list, and we also need to add a `bookId` prop so we can push the user to the correct route to update the review:

`client/src/components/ReviewList/index.js`

```
import { useHistory } from "react-router-dom";
import dayjs from "dayjs";

import Button from "../../components/Button";

function ReviewsList({ bookId, reviews, viewerId }) {
  const history = useHistory();

  return reviews.map(({ id, rating, reviewedOn, reviewer, text }) => (
    <div className="pt-10" key={id}>
      <div className="sm:flex sm:justify-between mb-4 sm:mb-0">
```

```
  {/* ... */}
  {viewerId === reviewer.id && (
    <div>
      <Button
        onClick={() => {
          history.push(`/book/${bookId}/review/${id}`);
        }}
        text="Update"
      />
    </div>
  )}
  <p>{text}</p>
</div>
));
}

export default ReviewsList;
```

The `ReviewList` component rendered in the `Book` page component must now also be updated to pass through the new props:

`client/src/pages/Book/index.js`

```
// ...

function Book() {
// ...

if (loading && !data) {
  content = <Loader centered />;
} else if (data?.book) {
// ...

content = (
  <div className="bg-white p-8 shadow-xl">
    {/* ... */}
    <div className="mt-8">
      {/* ... */}
      {reviews.results?.length ? (
        <div>
          <ReviewsList
```

```
        bookId={id}
        reviews={reviews.results}
        viewerId={viewer?.id || null}
      />
      {/* ... */}
    </div>
  ) : (
    <p className="italic mt-4">No reviews for this book yet!</p>
  )}
</div>
</div>
);
} else if (error) {
  content = <PageNotice text="Book not found!" />;
}

return <MainLayout>{content}</MainLayout>;
}

export default Book;
```

For the remainder of this section, we'll focus on reworking the `ReviewBook` page component so that it can serve the dual purposes of submitting new reviews and updating existing reviews. We have already designated a separate route to render the version of this page for updating reviews, but we'll need to make some adjustments to the user interface so that it's more applicable to that context. As a first step, if a user is updating their review, then existing values for the rating and the review content should be pre-populated in their respective fields so the user can see what they previously submitted. One way to get these values will be to fetch the individual existing review when the `ReviewBook` page component renders. And for that, we'll need another query:

`client/src/graphql/queries.js`

```
// ...

export const GetReview = gql`query GetReview($id: ID!) {
  review(id: $id) {
    id
    rating
    text
  }
}
```

```
`;  
// ...
```

Now we'll import the new `GetReview` query and the `UpdateReview` mutation into `ReviewBook`. We'll also need the `useQuery` hook from Apollo Client and the `useParams` hook from React Router to get the review's ID out of the route so we can use it as the variable value for `GetReview`. Lastly, we'll need the `Loader` component too so we can display it while the query data is fetched from the server.

Next, we'll call `useQuery` to fetch the review. We add the `skip` option to this query to tell Apollo Client that we don't want to bother fetching the query if the `reviewId` isn't available (because that would mean the component is being rendered to create a new review) or if for some reason the `viewer` isn't available. After the query is fetched, we use the review's `rating` and `text` fields to pre-populate the values of the form inputs. Lastly, we also pass the `UpdateReview` mutation into the `useMutation` hook:

`client/src/pages/ReviewBook/index.js`

```
import { useHistory, useParams } from "react-router-dom";  
import { useMutation, useQuery } from "@apollo/client";  
import { useState } from "react";  
  
import { CreateReview, UpdateReview } from "../../graphql/mutations";  
import { GetReview } from "../../graphql/queries";  
import { useAuth } from "../../context/AuthContext";  
import Button from "../../components/Button";  
import Loader from "../../components/Loader";  
// ...  
  
function ReviewBook() {  
  const { bookId, reviewId } = useParams();  
  // ...  
  
  const { loading, error } = useQuery(GetReview, {  
    variables: { id: reviewId },  
    skip: !reviewId || !viewer,  
    onCompleted: data => {  
      if (data?.review) {  
        setRating(data.review.rating);  
        setText(data.review.text);  
      }  
    }  
  }  
}
```

```
});  
  
const [createReview] = useMutation(CreateReview, {  
  onCompleted: () => {  
    history.push(`/book/${bookId}`);  
  }  
});  
const [updateReview] = useMutation(UpdateReview, {  
  onCompleted: () => {  
    history.push(`/book/${bookId}`);  
  }  
});  
  
// ...  
}  
  
export default ReviewBook;
```

Moving further down the component, we'll update the `handleSubmit` function so that it conditionally calls the `updateReview` mutation function if the `reviewId` is available:

*client/src/pages/ReviewBook/index.js*

```
// ...  
  
function ReviewBook() {  
  // ...  
  
  const handleSubmit = event => {  
    event.preventDefault();  
  
    if (reviewId) {  
      updateReview({  
        variables: {  
          input: {  
            id: reviewId,  
            rating: parseInt(rating),  
            text  
          }  
        }  
      }).catch(err => {  
        console.error(err);  
      }  
    }  
  };  
  return (  
    <Formik  
      initialValues={initialValues}  
      validationSchema={validationSchema}  
      onSubmit={handleSubmit}  
    >  
  );  
};
```

```
    });
} else {
  createReview({
    variables: {
      input: {
        bookId,
        rating: parseInt(rating),
        reviewerId: viewer.id,
        text
      }
    }
  }).catch(err => {
  console.error(err);
});
}
};

// ...
}

export default ReviewBook;
```

Lastly, we'll add some conditional rendering to the component to show the `Loader` when the existing review content is fetched and to render the heading text conditionally above the form. We'll account for any potential error states resulting from the `GetReview` query by including its `error` as a condition to check when rendering the `PageNotice`:

`client/src/pages/ReviewBook/index.js`

```
// ...

function ReviewBook() {
// ...

let content = null;

if (loading) {
  content = <Loader centered />;
} else if (viewer) {
  content = (
    <div className="bg-white p-8 shadow-xl">
      <h2 className="mb-8">
```

```
        {reviewId ? "Update Review" : "Create a New Review"}  
    </h2>  
    <form onSubmit={handleSubmit}>  
        {/* ... */}  
    </form>  
    </div>  
);  
} else if (error || viewerError) {  
    content = <PageNotice text="Something went wrong. Please try again!">  
    />;  
}  
  
return <MainLayout>{content}</MainLayout>;  
}  
  
export default ReviewBook;
```

Try out the new and improved `ReviewBook` component by updating a review that belongs to the currently authenticated user. After a successful mutation, it will behave exactly as the other version of the form does and redirect the user to the book page where they'll be able to see their updated review.

## Delete a Book Review

As the final piece of the CRUD puzzle for reviews, we'll implement the `deleteReview` mutation in the React application. As a first step, we'll add the new mutation to `mutations.js`:

`client/src/graphql/mutations.js`

```
// ...  
  
export const DeleteReview = gql`  
mutation DeleteReview($id: ID!) {  
    deleteReview(id: $id)  
}  
`;  
  
// ...
```

We'll add a button to trigger this mutation directly under the existing “Update” button for any review that belongs to the authenticated user. That means we'll need to import the `DeleteReview` mutation into the `ReviewList` component and update it with the following code:

client/src/components/ReviewList/index.js

```
import { useHistory } from "react-router-dom";
import { useMutation } from "@apollo/client";
import dayjs from "dayjs";

import { DeleteReview } from "../../graphql/mutations";
import Button from "../../components/Button";

function ReviewsList({ bookId, reviews, viewerId }) {
  const history = useHistory();

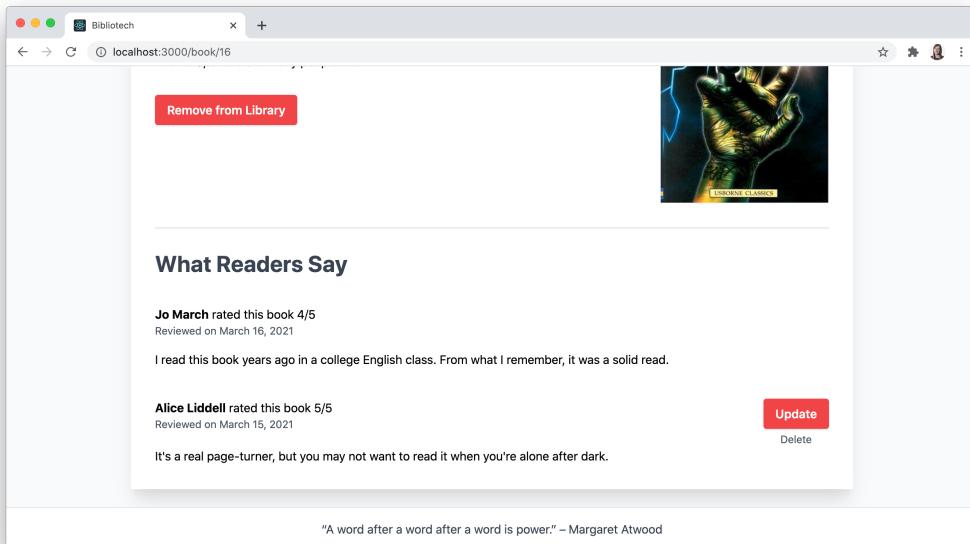
  const [deleteReview] = useMutation(DeleteReview);

  return reviews.map(({ id, rating, reviewedOn, reviewer, text }) => (
    <div className="pt-10" key={id}>
      <div className="sm:flex sm:justify-between mb-4 sm:mb-0">
        {/* ... */}
        {viewerId === reviewer.id && (
          <div>
            <Button
              className="block"
              onClick={() => {
                history.push(`/book/${bookId}/review/${id}`);
              }}
              text="Update"
            />
            <button
              className="block m-auto mt-1 text-sm text-gray-600
              hover:text-red-600"
              onClick={() => {
                deleteReview({ variables: { id } }).catch(err => {
                  console.log(err);
                });
              }}
            >
              Delete
            </button>
          </div>
        )}
      </div>
      <p>{text}</p>
```

```
        </div>
    )));
}

export default ReviewsList;
```

The new “Delete” button should render as follows:



To keep the user interface and Apollo Client cache in sync after a review is deleted, we'll have to use a different approach than we have before. When creating and updating reviews, we took the (somewhat heavy-handed) approach of refetching the book and review data from the GraphQL API when the Book page component reloads after a successful mutation. When deleting a review, the user will be able to take that action directly from the book page, and the deleted review should be removed from the list and the cache instantly when the mutation completes.

To accomplish this, we'll have to use Apollo Client's `cache.modify` method to extract the review reference from the applicable book object and also clean up the orphaned review object. To use this method, we'll call it from the mutation's `update` method (which has access to the `cache` object as its first parameter) and pass the `cache.modify` method an object that has an `id` property with the value of the `cache key` for the book we want to update, as well as a `fields` property that instructs Apollo Client how to update the data stored for the `reviews` field:

*client/src/components/ReviewList/index.js*

```
// ...

function ReviewsList({ bookId, reviews, viewerId }) {
  const history = useHistory();

  const [deleteReview] = useMutation(DeleteReview, {
    update: (cache, { data: { deleteReview } }) => {
      cache.modify({
        id: `Book:${bookId}`,
        fields: {
          reviews(existingReviewRefs, { readField }) {
            return existingReviewRefs.results.filter(
              reviewRef => deleteReview !== readField("id", reviewRef)
            );
          }
        });
      }
    }
  });

  // ...
}

export default ReviewsList;
```

Lastly, we'll use the `cache.evict` method to remove the review object from the cache too:

*client/src/components/ReviewList/index.js*

```
// ...

function ReviewsList({ bookId, reviews, viewerId }) {
  const history = useHistory();

  const [deleteReview] = useMutation(DeleteReview, {
    update: (cache, { data: { deleteReview } }) => {
      cache.modify({
        // ...
      });
      cache.evict({ id: `Review:${deleteReview}` });
    }
  });

  // ...
}

export default ReviewsList;
```

```
});  
// ...  
}  
  
export default ReviewsList;
```

Now the cache state will match the server and the user interface will be updated when the review is deleted. However, there is a lingering bug in the review list still. If the number of total reviews exceeds the page limit and we try deleting a few reviews, we'll notice some strange behavior with the pages of results that are displayed on the screen. At this point, we have finally bumped into one of the limitations of offset-based pagination—the paging window may become ambiguous when results are added to or removed in place from the current page. We could attempt to engineer some elaborate client-side workarounds to deal with this limitation, but requiring clients to jump through hoops to consume data from an API wouldn't be a very GraphQL-like approach to solving this problem.

Ultimately, to support the review deletion feature as designed (and support real-time updates to this list when new reviews are added with subscription operation in Chapter 10) the `reviews` field on the `Book` type should be adjusted to support cursor-based pagination. While JSON Server has served us well so far as a mocked REST API for our backing data source, it isn't well-suited to implementing cursor-based pagination. Working around these limitations to implement cursor-based pagination for the book reviews is outside the scope of this book. Nonetheless, this does serve as an important lesson in how product requirements may necessitate different pagination styles for different fields in a GraphQL API.

## Create a New Book with Authors

For our final mutation-powered feature, we'll add a mutation that allows users to submit new books to the Bibliotech catalogue. The form that we're going to build to create new books will have text input fields for the title, cover image URL, and summary, as well as a select field for the genre. The form will also include a text input for users to provide the name of the book author (or multiple authors, if applicable).

To make this experience as seamless as possible on the front-end, when a user submits a new book we'll also automatically create any authors that are associated with the book but that don't exist in `db.json` yet. Thinking through how to achieve this with our existing GraphQL API, when a user submits the form with the new book details, we would need to do the following:

1. Send a search query to the API for each of the author names listed in the authors input
2. If a result is returned for any of the authors names, then keep track of the author ID
3. For any author names that don't provide a search result, create each of those authors and then keep track of their IDs as well

4. Finally, send the `createBook` mutation with the existing author IDs and the new author IDs concatenated together as the `authorIds` argument value

In a REST API world, we may have to live with handling all of this business logic on the client. But we're building a GraphQL API, so a convoluted client-side scenario such as this is usually a good indication that a new mutation field can be added to simplify this process. What's more, shifting this logic to the server will make the process less error prone and more performant, as it will no longer be up to individual clients to ensure business rules are adhered to when creating a new book with authors and we can reduce the number of back-and-forth trips between the client and server to carry out the user's desired action.

Over in the `server` directory, our first step will be to add a new `createBookAndAuthors` mutation and a `CreateBookAndAuthorsInput` Input Object type to the type definitions:

`server/src/graphql/typeDefs.js`

```
import { gql } from "apollo-server-express";

const typeDefs = gql`  
# ...  
  
"""  
Provides data to create a book and any associated authors.  
"""  
input CreateBookAndAuthorsInput {  
  "The names of the authors who wrote the book. Non-existent authors will  
  be created."  
  authorNames: [String]  
  """  
    The URL of the book's cover image. Covers available via the Open  
    Library Covers API:  
  
    https://openlibrary.org/dev/docs/api/covers  
    """  
  cover: String  
  "A literary genre to which the book can be assigned."  
  genre: Genre  
  "A short summary of the book's content."  
  summary: String  
  "The title of the book."  
  title: String!  
}  
  
# ...
```

```
type Mutation {
  # ...
  "Creates a new book and any of its authors that do not yet exist."
  createBookAndAuthors(input: CreateBookAndAuthorsInput!): Book!
  # ...
}
;

export default typeDefs;
```

The data source method we need to help resolve this field will be somewhat complex because it will handle the server-side equivalent of what was outlined in the steps above. However, this code will still be more succinct than what would be required to implement the same logic on the client:

*server/src/graphql/dataSources/JsonServerApi.js*

```
// ...

class JsonServerApi extends RESTDataSource {
  // ...

  async createBookAndAuthors({ authorNames, ...args }) {
    let authorIds = [];

    if (authorNames?.length) {
      const authorSearchResults = await Promise.all(
        authorNames.map(authorName =>
          this.get(`/authors?name=${authorName}`))
      );
      const existingAuthors = authorSearchResults.flat();
      const existingAuthorIds = existingAuthors.map(author => author.id);
      authorIds.push(...existingAuthorIds);

      const authorsToCreate = existingAuthors.length
        ? authorNames.filter(
            authorName =>
              !existingAuthors.find(author => author.name === authorName)
          )
        : authorNames;

      const newAuthorIds = [];
```

```
    if (authorsToCreate.length) {
      const newAuthors = await Promise.all(
        authorsToCreate.map(name => this.createAuthor(name))
      );
      newAuthors.forEach(newAuthor => {
        newAuthorIds.push(newAuthor.id);
      });
    }
    authorIds.push(...newAuthorIds);
  }

  return this.createBook({ authorIds, ...args });
}

// ...
}

export default JsonServerApi;
```

Next, we'll create a resolver for the new `createBookAndAuthors` mutation field:

*server/src/graphql/resolvers.js*

```
// ...

const resolvers = {
  // ...
  Mutation: {
    // ...
    createBookAndAuthors(root, { input }, { dataSources }, info) {
      return dataSources.jsonServerApi.createBookAndAuthors(input);
    },
    // ...
  }
};

export default resolvers;
```

Back over on the client, we can add a `CreateBookAndAuthors` mutation now:

client/src/graphql/mutations.js

```
// ...  
  
export const CreateBookAndAuthors = gql`  
  mutation CreateBookAndAuthors($input: CreateBookAndAuthorsInput!) {  
    createBookAndAuthors(input: $input) {  
      id  
      authors {  
        name  
      }  
      cover  
      genre  
      title  
    }  
  }  
`;  
  
// ...
```

Next, we'll create a new page component directory called `NewBook` in `client/src/pages` to render the form that will be responsible for capturing new book submissions. In its `index.js` file, we'll add the following initial code for the form and its fields:

client/src/pages/NewBook/index.js

```
import { useHistory } from "react-router-dom";  
import { useMutation } from "@apollo/client";  
import { useState } from "react";  
  
import { CreateBookAndAuthors } from "../../graphql/mutations";  
import { useAuth } from "../../context/AuthContext";  
import Button from "../../components/Button";  
import MainLayout from "../../components/MainLayout";  
import PageNotice from "../../components/PageNotice";  
import Select from "../../components/Select";  
import TextInput from "../../components/TextInput";  
  
function NewBook() {  
  const [title, setTitle] = useState("");  
  const [authors, setAuthors] = useState([]);  
  const [cover, setCover] = useState("");  
  const [genre, setGenre] = useState("");
```

```
const [summary, setSummary] = useState("");
const [titleError, setTitleError] = useState();

const { viewer, error: viewerError } = useAuth();
const history = useHistory();

const handleSubmitBook = async event => {
  event.preventDefault();
  // Check for an empty title field and send the mutation here...
};

let content = null;

if (viewer) {
  content = (
    <div className="bg-white p-8 shadow-xl">
      <h2 className="mb-8">Create a New Book</h2>
      <form onSubmit={handleSubmitBook}>
        <TextInput
          className="mb-6 w-full sm:w-3/5 md:w-1/2"
          error={titleError}
          id="title"
          inputWidthClass="w-full"
          label="Title *"
          name="title"
          onChange={event => {
            setTitle(event.target.value);
          }}
          placeholder="Enter the book's title"
          value={title}
        />
        <TextInput
          className="mb-6 w-full sm:w-3/5 md:w-1/2"
          id="cover"
          inputWidthClass="w-full"
          label="Cover Image URL"
          name="cover"
          onChange={event => {
            setCover(event.target.value);
          }}
          placeholder="Provide an URL for the book's cover image"
          value={cover}
        />
    </div>
  );
}

export default function CreateBook() {
  return (
    <div>
      {content}
    </div>
  );
}
```

```
    />
  <TextInput
    className="mb-6 w-full sm:w-3/5 md:w-1/2"
    id="author"
    inputWidthClass="w-full"
    label="Author (separate multiple authors with commas)"
    name="author"
    onChange={event => {
      setAuthors(event.target.value);
    }}
    placeholder="Enter the author's full name"
    value={authors}
  />
  <TextInput
    className="mb-6 w-full sm:w-3/5 md:w-1/2"
    id="summary"
    inputWidthClass="w-full"
    label="Summary"
    name="summary"
    onChange={event => {
      setSummary(event.target.value);
    }}
    placeholder="Provide a short summary of the book's content"
    value={summary}
  />
  <Select
    className="mb-6 w-full sm:w-3/5 md:w-1/2"
    label="Genre"
    onChange={event => {
      setGenre(event.target.value);
    }}
    options={[
      { text: "Choose one...", value: "" },
      { text: "Adventure", value: "ADVENTURE" },
      { text: "Children", value: "CHILDREN" },
      { text: "Classics", value: "CLASSICS" },
      {
        text: "Comic Book / Graphic Novel",
        value: "COMIC_GRAPHIC_NOVEL"
      },
      {
        text: "Detective / Mystery",
        value: "DETECTIVE_MYSTERY"
      }
    ]}
  />
```

```
        value: "DETECTIVE_MYSTERY"
    },
    { text: "Dystopia", value: "DYSTOPIA" },
    { text: "Fantasy", value: "FANTASY" },
    { text: "Horror", value: "HORROR" },
    { text: "Humor", value: "HUMOR" },
    { text: "Non-Fiction", value: "NON_FICTION" },
    {
        text: "Science Fiction",
        value: "SCIENCE_FICTION"
    },
    { text: "Romance", value: "ROMANCE" },
    { text: "Thriller", value: "THRILLER" },
    { text: "Western", value: "WESTERN" }
]}
selectWidthClass="w-full sm:w-3/4"
value={genre}
/>
<div className="flex items-center">
    <Button className="mr-2 mt-4" primary text="Submit"
        type="submit" />
</div>
</form>
</div>
);
} else if (viewerError) {
    content = <PageNotice text="Something went wrong. Please try again!" />;
}

return <MainLayout>{content}</MainLayout>;
}

export default NewBook;
```

Next, we'll use the `useMutation` hook to configure the `CreateBookAndAuthors` mutation. In its `onCompleted` option, we'll push the user to the route of the newly created book:

*client/src/components/NewBook/index.js*

```
// ...
```

```
function NewBook() {
  // ...

  const [createBookAndAuthors, { error: mutationError }] = useMutation(
    CreateBookAndAuthors,
    {
      onCompleted: ({ createBookAndAuthors: { id } }) => {
        history.push(`/book/${id}`);
      }
    }
  );

  const handleSubmitBook = async event => {
    event.preventDefault();
    // Check for an empty title field and send the mutation here...
  };

  let content = null;

  if (viewer) {
    // ...
  } else if (mutationError || viewerError) {
    content = <PageNotice text="Something went wrong. Please try again!" />;
  }

  return <MainLayout>{content}</MainLayout>;
}

export default NewBook;
```

We must also update the `handleSubmitBook` function to call the mutation. Because the title is a required field, we first check if a `title` value exists and set an error on the field if it doesn't:

`client/src/components/NewBook/index.js`

```
// ...

function NewBook() {
  // ...

  const handleSubmitBook = async event => {
```

```
event.preventDefault();
setTitleError(null);

if (!title) {
  setTitleError("This field is required");
  return;
}

createBookAndAuthors({
  variables: {
    input: {
      authorNames: authors ? authors.split(/\s*,\s*/) : [],
      title,
      ...(cover && { cover }),
      ...(genre && { genre }),
      ...(summary && { summary })
    }
  }
}).catch(err => {
  console.error(err);
});
};

// ...
}

export default NewBook;
```

To view our new form, we'll render it at the /book/new route:

*client/src/router/index.js*

```
import { Switch } from "react-router";

// ...
import NewBook from "../pages/NewBook";
// ...

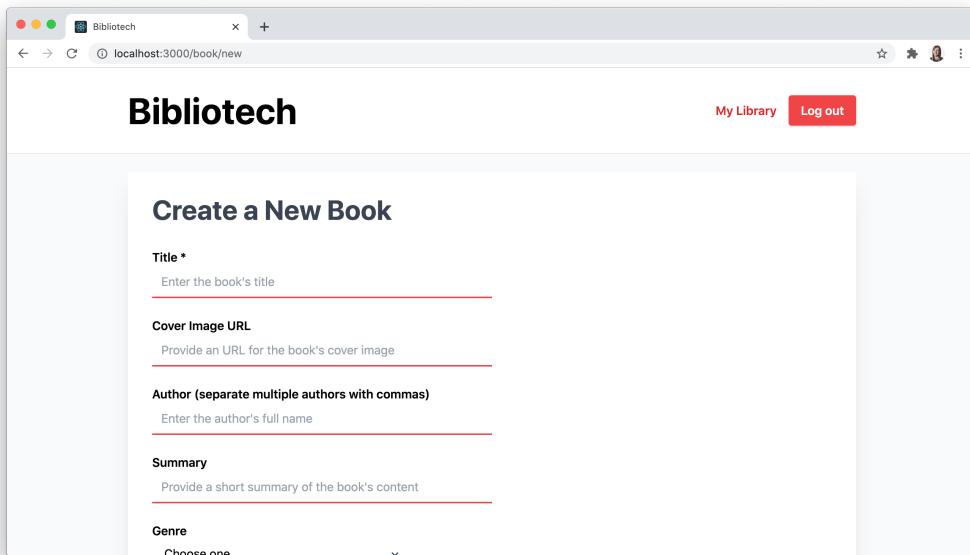
export function Routes() {
  return (
    <Switch>
      {/* ... */}
```

```
        <PrivateRoute exact path="/book/new" component={NewBook} />
        <PublicRoute exact path="/book/:id" component={Book} />
        {/* ... */}
    </Switch>
);

}

export default Routes;
```

The completed “Create a New Book” form should render like this:



As a final step, let’s add an “Add Book” link to the NavBar component so authenticated users can easily access this page from anywhere in the application:

*client/src/components/NavBar/index.js*

```
// ...

function NavBar() {
// ...

return (
<header className="bg-white border-b border-gray-200 border-solid">
```

```
<div className="flex items-center justify-between mx-auto  
    max-w-screen-lg px-8 py-8 w-full">  
    {/* ... */}  
    <div className="flex items-center sm:justify-end mt-2 sm:mt-0">  
        {isAuthenticated() && (  
            <>  
                <Link to="/home">  
                    <span className="font-semibold mr-4 text-sm sm:text-base  
                        text-red-600">  
                        My Library  
                    </span>  
                </Link>  
                <Link to="/book/new">  
                    <span className="font-semibold mr-4 text-sm sm:text-base  
                        text-red-600">  
                        Add Book  
                    </span>  
                </Link>  
            </>  
        )}  
        {/* ... */}  
    </div>  
    </div>  
    </header>  
);  
  
export default NavBar;
```

To wrap up this chapter, try creating a new book with new authors to confirm that the form is functioning as expected.

## Summary

In this chapter, we learned about some of Apollo Client's more advanced features, including how to interact with the normalized cache when fetching additional pages of results and when running mutations. We paginated lists of books on the index and home pages, and also paginated lists of reviews on the book pages. We created mutations for adding and removing books from a library, as well as creating, updating, and deleting reviews. We also adjusted our API to accommodate a product requirement to allow users to create new books and any applicable, non-existent authors from a single form.

We have now built out most of the features for Bibliotech's React application, but we still have one more finishing touch to add. In the next chapter, we'll learn how to add new reviews in real-time to books pages using GraphQL subscriptions.

# Real-time Updates with Subscriptions

In this chapter, we will:

- Configure a GraphQL server to support subscription operations
- Add a subscription to the GraphQL API that supports pushing new reviews for a single book to subscribed clients
- Create custom Apollo Links to handle requests to a WebSocket endpoint and authentication errors that are returned responses
- Subscribe to updates from a GraphQL API after an initial query is sent from a client

## Support a Subscription for New Reviews

The first nine chapters of this book focused on making requests to a GraphQL API using the query and mutation root operation types. In this concluding chapter, we will finally turn our attention to the third root operation type outlined in the GraphQL specification. Fields queried using the subscription root operation type will allow clients to receive data from the API via long-lived requests that fetch new data as it becomes available on the server.

Specifically, we will create a `reviewAdded` field on the root `Subscription` type in the schema to allow Bibliotech's React application to add new reviews to the applicable book pages in real-time. Unlike the query and mutation operations that are commonly handled using HTTP as a transport mechanism, GraphQL subscription operations are often handled using [WebSockets](#) to persist an open connection between the client and server. That means we'll need to do some additional configuration on the server and client sides to add support for subscriptions. But before we jump into how to add subscription support to a GraphQL API, we should pause to consider why we would choose to support any subscription operations in the first place.

Subscription operations are best used for data that changes frequently and incrementally, and also when keeping that data up-to-date in an interface is essential to user experience. GraphQL subscriptions can be a very convenient tool to reach for to support client application features that have these requirements, but there are important trade-offs to consider. For example,

horizontally scaling servers that respond to WebSocket requests will be more complicated than scaling servers that only respond to HTTP requests because these connections are stateful and each client needs to remain bound to a specific instance of the server. Additionally, WebSocket connections may have battery implications for mobile devices. So for less frequent updates, it may be worth considering whether another solution such as periodic polling may provide a better answer instead.

Using a subscription operation to receive real-time book review updates may seem like overkill based on the previous criteria. But for educational purposes, we'll proceed with adding the `reviewAdded` subscription so we can see what needs to be wired up in the server and client applications to support this operation (because this would be an awfully short chapter if we didn't!).

Our first step will be to install some packages that will help us handle WebSocket connections and subscription operations on the server. Historically, the `subscriptions-transport-ws` package has been the go-to option for setting up subscriptions directly with Apollo Server 2.x. However, that package is no longer maintained and [subscription support in Apollo Server 3](#) will likely work quite differently when it eventually lands, so we will opt for using the `graphql-ws` library instead. We can use this package alongside Apollo Server Express, so from the client's perspective, the experience of consuming data via the `subscription` root operation type will be similar to using Apollo Server's built-in subscription support. Let's install `graphql-ws` in the `server` directory now, as well as a WebSocket library for Node.js called `ws`:

```
npm i graphql-ws@4.3.1 ws@7.4.4
```

Next, we'll update the imports at the top of the main `index.js` file in `server/src`:

`server/src/index.js`

```
import http from "http";

import { ApolloServer, makeExecutableSchema } from "apollo-server-express";
import { applyMiddleware } from "graphql-middleware";
import { useServer } from "graphql-ws/lib/use/ws";
import cors from "cors";
import express from "express";
import expressJwt from "express-jwt";
import ws from "ws";

// ...
```

Note that we import the `http` module from Node.js because we need access to the HTTP server powering the GraphQL API directly now, rather than allowing Express to create this server instance under the hood for us. To do that, instead of calling `app.listen` at the bottom of the

file we'll call `http.createServer` and pass the Express app into it. Then we'll call the `listen` method on the `httpServer` object and create the WebSocket server at `/graphql` in its callback first and then call the `useServer` function from `graphql-ws`, passing it the executable schema and the new `wsObject`:

*server/src/index.js*

```
// ...

server.applyMiddleware({ app, cors: false });
const httpServer = http.createServer(app);

httpServer.listen(port, () => {
  const wsServer = new ws.Server({ server: httpServer, path: "/graphql" });
  useServer({ schema: schemaWithPermissions }, wsServer);

  console.log(
    `Server ready at http://localhost:${port}${server.graphqlPath}`
  );
  console.log(
    `Subscriptions ready at ws://localhost:${port}${wsServer.options.path}`
  );
});
```

For subscription operations, the `JsonServerApi` data source won't be added to the resolver context automatically, so we'll need to use the `context` option in the object argument passed into `useServer` to manually set this data source on the context:

*server/src/index.js*

```
// ...

httpServer.listen(port, () => {
  const wsServer = new ws.Server({ server: httpServer, path: "/graphql" });
  useServer(
    {
      schema: schemaWithPermissions,
      context: ctx => {
        const jsonServerApi = new JsonServerApi();
        jsonServerApi.initialize({ context: ctx, cache: undefined });
        return { dataSources: { jsonServerApi } };
      }
    },
  );
```

```
    wsServer
);
// ...
});
```

Behind the scenes, GraphQL subscriptions are typically supported by some kind of pub/sub mechanism so that relevant subscription-related messages may be published at various points during runtime (often in a mutation resolver) and then received by subscription resolvers. There's no built-in pub/sub feature in Apollo Server or graphql-ws, so we'll need to install another package for that as well:

```
npm i graphql-subscriptions@1.2.1
```

This library will provide us with a basic, in-memory pub/sub implementation that will work well for development purposes. In production, however, we'd likely want to upgrade to [a more advanced pub/sub solution](#) that uses the PubSubEngine abstract class bundled in this package. Now we're ready to add the `Subscription` type with a `reviewAdded` field to our type definitions:

`server/src/graphql/typeDefs.js`

```
import { gql } from "apollo-server-express";

const typeDefs = gql` 
# ...

type Subscription {
  reviewAdded(bookId: ID!): Review
}
`;

export default typeDefs;
```

We add the `bookId` as a field argument above because when a new review is added we want to make sure any user viewing a book page only receives updates about new reviews for that specific book (rather than every single book in the Bibliotech catalog).

Next, we'll instantiate a new `PubSub` object in `resolvers.js` and also create a constant for a `REVIEW_ADDED` topic that we will use to capture all messages related to new reviews:

*server/src/graphql/resolvers.js*

```
import { PubSub } from "graphql-subscriptions";

import DateTimeType from "./scalars/DateTimeType.js";
import RatingType from "./scalars/RatingType.js";

const pubsub = new PubSub();
const REVIEW_ADDED = "REVIEW_ADDED";

// ...
```

Whenever a new review is added, we'll publish a message to the REVIEW\_ADDED topic with a payload that contains the review data. To do that, we'll call the publish method on the pubsub object we previously instantiated, passing it the topic name and the review payload:

*server/src/graphql/resolvers.js*

```
// ...

const resolvers = {
  // ...
  Mutation: {
    // ...
    async createReview(root, { input }, { dataSources }, info) {
      const review = await dataSources.jsonServerApi.createReview(input);
      pubsub.publish(REVIEW_ADDED, { reviewAdded: review });
      return review;
    },
    // ...
  }
};

export default resolvers;
```

Next, we must add a resolver for the reviewAdded field. This resolver will call the `asyncIterator` method on the same pubsub object and pass this method the constant for the REVIEW\_ADDED topic so it knows what topic is should listen to:

*server/src/graphql/resolvers.js*

```
// ...
```

```
const resolvers = {
  // ...
  Mutation: {
    // ...
  },
  Subscription: {
    reviewAdded: {
      subscribe(root, args, context, info) {
        return pubsub.asyncIterator([REVIEW_ADDED]);
      }
    }
  }
};

export default resolvers;
```

Before moving on, let's pause and consider what the above code does. Whenever a message about any new review is posted to the REVIEW\_ADDED topic, this data will be pushed down to subscribed clients. However, we only want to send updates to clients if the review belongs to the book that they are currently viewing. To tame this potential firehose, we can use the `withFilter` function provided by `graphql-subscriptions` to set a criterion for what review updates will be shared with a given subscribed client. Here, this filter will be set based on the `bookId` value provided in the operation document initially sent by the client. The `withFilter` function takes the resolver function we previously defined as the first argument, and as a second argument, it takes a function that returns a boolean to indicate if the published message applies to the subscribed client:

`server/src/graphql/resolvers.js`

```
import { PubSub, withFilter } from "graphql-subscriptions";

// ...

const resolvers = {
  // ...
  Subscription: {
    reviewAdded: {
      subscribe: withFilter(
        (root, args, context, info) => {
          return pubsub.asyncIterator([REVIEW_ADDED]);
        },
        (payload, variables, context, info) => {
          return payload.reviewAdded.bookId === parseInt(variables.bookId);
        }
      )
    }
};

export default resolvers;
```

```

        }
    )
}
};

export default resolvers;

```

With this code in place, our React application will now be able to send a subscription operation for new reviews to the Bibliotech GraphQL API.

## Connect to the WebSocket Endpoint with Apollo Link

Jumping back over to the client side, we also have some additional configuration to do to be able to send subscription operations including the `reviewAdded` field to the new WebSocket-powered endpoint. When we originally configured Apollo Client in Chapter 8 we instantiated an `HttpLink` and set that link as the `link` option in the `ApolloClient` constructor. While it may have seemed that there wouldn't be much else for us to do with the `link` property beyond that, in reality, it enables a very powerful feature of Apollo Client called [Apollo Links](#). So to send requests to the WebSocket endpoint, we'll need to use a custom Apollo Link.

Apollo Links allow us to modify the way we send requests to a GraphQL API. The most important thing to know about Apollo Links is that they allow us to chain together the units of work we want to perform before getting the result of a GraphQL operation. These units of work can be composed together so that the first “link” in the chain operates on the original GraphQL operation object and the links that follow work on the output of the link that precedes it.

The last link in the composed chain will be a *terminating link*, and for our purposes, this link will send a network request to the server to fetch a result for the GraphQL operation. This is what the `HttpLink` currently does for us:



Note that a terminating link doesn't necessarily need to fetch data from a server if there is some other way to obtain the desired execution result. A standard Apollo Client set-up will use the `HttpLink` because it creates the typical terminating link for fetching data from a GraphQL endpoint over an HTTP connection. To allow a client application to send subscription operations to the WebSocket endpoint, we will need to create a terminating link for this purpose too and then conditionally send requests to the API depending on what kind of root operation type the

operation document contains. To do that, we can use the `createClient` function provided in the same `graphql-ws` library that we used on the server. Let's install `graphql-ws` in the `client` directory now:

```
npm i graphql-ws@4.3.2
```

The `graphql-ws` documentation provides a template for creating an Apollo Link using the `createClient` function, so we'll create a new `links` directory in `client/src/graphql` and copy and past that template into a new `WebSocketLink.js` file in there:

`client/src/graphql/links/WebSocketLink.js`

```
import { ApolloLink, Observable } from "@apollo/client";
import { createClient } from "graphql-ws";
import { print } from "graphql";

class WebSocketLink extends ApolloLink {
  constructor(options) {
    super();
    this.client = createClient(options);
  }

  request(operation) {
    return new Observable(sink => {
      return this.client.subscribe(
        { ...operation, query: print(operation.query) },
        {
          next: sink.next.bind(sink),
          complete: sink.complete.bind(sink),
          error: err => {
            if (err instanceof Error) {
              return sink.error(err);
            }

            if (err instanceof CloseEvent) {
              return sink.error(
                new Error(
                  `Socket closed with event ${err.code} ${err.reason} ||
                  ""`);
            }
          };
        });
    });
  }
}
```

```
        return sink.error(
          new Error(err.map(({ message }) => message).join(", "))
        );
      }
    );
  );
}

export default WebSocketLink;
```

When we create a `WebSocketLink` using this class, we will need to provide an `url` property in the object passed into it as an argument. The value or the `url` property will be retrieved from a new `REACT_APP_SUBSCRIPTIONS_ENDPOINT` variable that we'll create in the client's `.env` file:

`client/.env`

```
REACT_APP_GRAPHQL_ENDPOINT=http://localhost:3000/graphql
REACT_APP_SUBSCRIPTIONS_ENDPOINT=ws://localhost:3000/graphql
```

To use the Create React App proxy to establish a WebSocket connection, we'll need a more advanced proxy configuration than what we currently have in the client's `package.json` file. To configure the proxy manually, we'll first remove the `"proxy": "http://localhost:4000"` line from the client's `package.json` file and then install this package in the `client` directory:

```
npm i http-proxy-middleware@1.1.0
```

To get direct access to the Express app that powers Create React App's development server so that we can add custom proxy middleware, we'll create a `setupProxy.js` file in `client/src` with the following code:

`client/src/setupProxy.js`

```
const { createProxyMiddleware } = require("http-proxy-middleware");

const target = "http://localhost:4000";

module.exports = function (app) {
  app.use(
    "/graphql",
    createProxyMiddleware("/graphql", { target, ws: true })
}
```

```
    );  
};
```

You can read more about proxy configurations in the [Create React App documentation](#).

We don't need to import the `setupProxy.js` file anywhere because it will be registered automatically when the React application's development server starts. Now we're ready to set up a `WebSocketLink` for Apollo Client. We'll begin by importing that class at the top of the `apollo.js` file and then instantiating a new WebSocket-powered link and setting it as the `wsLink` variable. Similarly, we will now declare an `httpLink` variable and set its value as an instantiated `HttpLink` object instead of doing this directly in the object passed into the `ApolloClient` constructor:

`client/src/graphql/apollo.js`

```
import { ApolloClient, HttpLink, InMemoryCache } from "@apollo/client";  
  
import typePolicies from "./typePolicies";  
import WebSocketLink from "./links/WebSocketLink";  
  
const httpLink = new HttpLink({  
  uri: process.env.REACT_APP_GRAPHQL_ENDPOINT  
});  
  
const wsLink = new WebSocketLink({  
  url: process.env.REACT_APP_SUBSCRIPTIONS_ENDPOINT  
});  
  
// ...
```

We now have two different terminating links at our disposal, but we can only use one at a time at the end of an Apollo Link chain. To get around this constraint, we can use the `split` function provided by Apollo Client. The first argument to this function is a function that must return `true` or `false` and that takes the operation object as a parameter. The second argument is the link to use if the previous function returns `true`. The final (optional) argument is the link to use if the function returns `false`. For our purposes, we will write a test function to check if the root operation type is a subscription and use the `wsLink` if it is, otherwise, the `httpLink` will be used. We'll also need the help of the `getMainDefinition` utility function from Apollo Client to get the query definition in the test function:

client/src/graphql/apollo.js

```
import { ApolloClient, HttpLink, InMemoryCache, split } from
  "@apollo/client";
import { getMainDefinition } from "@apollo/client/utilities";

// ...

const link = split(
  ({ query }) => {
    const definition = getMainDefinition(query);
    return (
      definition.kind === "OperationDefinition" &&
      definition.operation === "subscription"
    );
  },
  wsLink,
  httpLink
);

const client = new ApolloClient({
  cache: new InMemoryCache({ typePolicies }),
  connectToDevTools: process.env.NODE_ENV === "development",
  link
});

export default client;
```

With this code in place, Apollo Client is now equipped to send subscription operations to the provided WebSocket endpoint while continuing to send query and mutation operations to the HTTP endpoint.

## Optional: Add an Apollo Link to Handle Expired JWTs

Now that we have a deeper understanding of how Apollo Link works, we can add another link to the chain before the terminating link that will enhance the error-handling experience when a user triggers an action that requires authentication after their JWT has expired. This link will first check if any errors were received in the response, then check if one of those errors was a `Not Authorised!` error from GraphQL Shield, then check if the user's JWT has passed its expiration time, and finally redirect the user back to the `/login` route using `history.push` if they need to re-authenticate.

When we create the custom error link, we won't be able to access the `history` object using the `useHistory` hook because we can only call hooks inside of React components or other hooks. To get around this limitation, we can install the `history` library to create a custom history object to use instead of the one that's automatically available with React Router's `BrowserRouter`. We'll begin by installing the package in the `client` directory:

```
npm i history@4.10.1
```

In `client/src/router/index.js`, we can now create and export the custom history object:

`client/src/router/index.js`

```
import { createBrowserHistory } from "history";

// ...

export const history = createBrowserHistory();
// ...
```

Next, we'll update the main `index.js` file for the client so that the router imports and uses the new `history` object. Because we can't use a custom history object with the `BrowserRouter`, we'll have to use the generic `Router` component from React Router instead now:

`client/src/index.js`

```
import { ApolloProvider } from "@apollo/client";
import { Router } from "react-router-dom";
import ReactDOM from "react-dom";

import { AuthProvider } from "./context/AuthContext";
import { history, Routes } from "./router";
import client from "./graphql/apollo";

import "./index.css";

function App() {
  return (
    <ApolloProvider client={client}>
      <AuthProvider>
        <Router history={history}>
          <Routes />
        </Router>
      </AuthProvider>
    </ApolloProvider>
  );
}

export default App;
```

```
        </AuthProvider>
      </ApolloProvider>
    );
}

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

Now we can add a new link to handle API errors resulting from expired JWTs. Apollo Client has a built-in `onError` link that we can use to intercept both GraphQL and network errors in a response to do custom handling. To use this link, we'll add an `authErrorLink.js` file in the `client/src/graphql/links` directory that we created in the last section and import `onError` from Apollo Client. Then we'll set up the callback passed into the `onError` link to check for a GraphQL error with a `Not Authorised!` message, and if that error is present, we'll check for an expired JWT, and then redirect the user to the `/login` route if needed:

`client/src/graphql/links/authErrorLink.js`

```
import { onError } from "@apollo/client/link/error";

import { history } from "../../router";

const authErrorLink = onError(({ graphQLErrors }) => {
  if (graphQLErrors) {
    const notAuthorizedError = graphQLErrors.find(
      error => error.message === "Not Authorised!"
    );

    if (notAuthorizedError) {
      const expiresAt = localStorage.getItem("token_expires_at");
      const isAuthenticated = expiresAt
        ? new Date().getTime() < expiresAt
        : false;

      if (!isAuthenticated) {
        history.push("/login");
      }
    }
  }
});

export default authErrorLink;
```

Lastly, we can insert the new error link before the terminating link by concatenating the `httpLink` onto the `authErrorLink` by calling the `concat` method on the link object:

`client/src/graphql/apollo.js`

```
// ...  
  
import authErrorLink from "./links/authErrorLink";  
// ...  
  
const link = split(  
  ({ query }) => {  
    const definition = getMainDefinition(query);  
    return (  
      definition.kind === "OperationDefinition" &&  
      definition.operation === "subscription"  
    );  
  },  
  wsLink,  
  authErrorLink.concat(httpLink)  
);  
  
// ...
```

Note that we don't need to concatenate the `authErrorLink` and the `wsLink` together because no protected fields are queryable through the subscription root operation type. To test out the new link, try temporarily shortening the expiration time of the JWT created in the `login` method in the `JsonServerApi` data source to be a few seconds, then try sending a mutation from the React application user interface after the JWT has expired, and confirm that the user is redirected to the login page.

## Add a Subscription Operation to the Client

Now that our GraphQL API has a `Subscription` type with a `reviewAdded` field, we have a pub/sub implementation in place, and our client is capable of sending subscription operations to the WebSocket endpoint, we can finally update the Book page component to use a subscription for new reviews. Let's create a new `subscriptions.js` file in `client/src/graphql` to house all of our subscriptions operations and then add the following code to that file:

client/src/graphql/subscriptions.js

```
import { gql } from "@apollo/client";

import { fullReview } from "./fragments";

export const ReviewAdded = gql`  
subscription ReviewAdded($bookId: ID!) {  
  reviewAdded(bookId: $bookId) {  
    ...fullReview  
  }  
}  
${fullReview}  
`;
```

There are two different ways that we can use this subscription operation with Apollo Client. The first is to use the `useSubscription` hook to initiate the connection with the server. The second is to use the `subscribeToMore` function returned from a query operation (much like the `fetchMore` function we previously used for pagination). The `subscribeToMore` function will be the right choice for us because we want to subscribe to new review updates for a book after the initial `GetBook` operation runs. To set up the subscription, we'll destructure `subscribeToMore` from the `GetBook` query result and call that function in the `useEffect` hook so that the subscription begins once the component mounts:

client/src/pages/Book/index.js

```
import { useEffect } from "react";
// ...

// ...
import { ReviewAdded } from "../../graphql/subscriptions";
// ...

function Book() {
// ...

  const { data, error, fetchMore, loading, subscribeToMore } = useQuery(
    GetBook,
    {
      variables: { id, reviewsLimit, reviewsPage: 1 },
      fetchPolicy: "cache-and-network",
      nextFetchPolicy: "cache-first"
    }
}
```

```
);

// ...

useEffect(() => {
  const unsubscribe = subscribeToMore({
    document: ReviewAdded,
    variables: { bookId: id }
  });
  return () => unsubscribe();
});

// ...
}
```

As we receive new review data from the server, we'll need to manually update the review list for the book too. To make that happen, we'll create a `updateAddNewReviewToList` function in `updateQueries.js`. This function will take the previous state of the book and the new review object and then merge that review data at the beginning of the list in the `reviews` field:

`client/src/utils/updateQueries.js`

```
// ...

export function updateAddNewReviewToList(previousResult, subscriptionData) {
  if (!subscriptionData.data) {
    return previousResult;
  }
  const newReview = subscriptionData.data.reviewAdded;

  return {
    book: {
      ...previousResult.book,
      reviews: {
        __typename: previousResult.book.reviews.__typename,
        results: [newReview, ...previousResult.book.reviews.results],
        pageInfo: previousResult.book.reviews.pageInfo
      }
    }
  };
}
```

```
// ...
```

Now we'll import the `updateViewerHasInLibrary` function into the `Book` page component file and use it with the `updateQuery` property passed into the object argument of the `subscribeToMore` function:

*client/src/pages/Book/index.js*

```
// ...
import {
  updateAddNewReviewToList,
  updateViewerHasInLibrary
} from "../../utils/updateQueries";
// ...

function Book() {
// ...

useEffect(() => {
  const unsubscribe = subscribeToMore({
    document: ReviewAdded,
    variables: { bookId: id },
    updateQuery: (previousResult, { subscriptionData }) =>
      updateAddNewReviewToList(previousResult, subscriptionData)
  });
  return () => unsubscribe();
});

let content = null;

// ...
}
```

With this code in place, try adding a new review in one browser tab with the relevant book page opened in another browser tab. The new review will now be automatically added to the page in real-time and without refreshing.

## Summary

Congratulations! You made it to the end of our extensive journey through the inner workings of GraphQL and Apollo. The final chapter covered off how to handle subscription operations on both

the server and client sides, which was the only root type operation we hadn't yet explored. We also saw how an Apollo Link can be used as a powerful tool for processing GraphQL requests before they leave the client as well as the responses that are returned from the server. If you're eager to continue GraphQL learning journey from here, be sure to check out Appendix A for a curated list of additional resources.

# Appendix A: GraphQL Resources

## Background

When first learning about GraphQL, the [official GraphQL website](#) maintained by The GraphQL Foundation is a great place to start. This website contains tutorials and links to third-party GraphQL resources to help you get started. You can also find current and past versions of the [GraphQL specification](#) here. At the time this book was written, the [June 2018 version of the specification](#) is the latest release.

For historical background on how GraphQL came into being at Facebook and the events leading up to its public release, you may be interested in watching [GraphQL: The Documentary](#). And if GraphQL has piqued your interest in graph theory, then [A Gentle Introduction To Graph Theory](#) by Vaidehi Joshi is a good primer.

## Apollo Resources

Apollo maintains extensive documentation on its open-source libraries and products. These sections are most relevant to this book:

- [Apollo Server](#)
- [Apollo Client \(React\)](#)
- [Apollo Studio Explorer](#)

Apollo also has beginner-level, interactive tutorials called [Odyssey](#) that can supplement what you learn throughout this book. For more video-based content covering a variety of topics related to GraphQL and Apollo, check out [Apollo on Twitch](#) and [GraphQL Summit](#). And be sure to bookmark the [Apollo Blog](#).

## Performance

When deploying a GraphQL API into a production environment, performance is an essential consideration. First, the API should be protected from malicious queries by considering measures such as [limiting query depth](#). For a more advanced implementation, query cost analysis can be used to tally up the “cost” of each field in a query document and reject queries that exceed a maximum budget (either on a per query basis or as a part of a rate limiting strategy). Query cost analysis can be quite nuanced and context-dependent, but there are [example query cost plugins](#) available as well as an [academic paper on the subject here](#).

Another important consideration is [solving the N+1 problem](#) that GraphQL queries may create by using [data loaders](#) for field resolvers. And despite some common misconceptions about GraphQL query responses being difficult to cache, there are proven caching strategies that work for GraphQL APIs. The [Caching & GraphQL: Setting the Story Straight talk](#) from by Marc-André Giroux at GraphQL Summit 2019 provides a good introduction, as well as the [Apollo Server documentation on Caching](#) and [Automatic Persisted Queries](#).

## Testing

For a quick primer on client-side and server-side testing with Apollo, check out the [Testing GraphQL talk](#) from Jake Dawkins from GraphQL Summit 2018. The Apollo documentation also has an entry on [testing React components](#) and there is a detailed post on the Apollo blog by Khalil Stemmler called [Testing Apollo Client Applications](#). You can also find information on how to do [mocking](#) and [integration testing](#) with Apollo Server in the documentation.

## Authentication

The GraphQL API authentication and authorization system demonstrated in this book is largely for educational purposes and not suitable for a production environment. For many cases, you would likely want to investigate options for using a third-party authentication service (or hire a team of experienced engineers to design a bespoke authentication system for you). To make an informed decision about how you implement authentication and authorization for a GraphQL API, it’s a good idea to familiarize yourself with general auth-related pitfalls and best practices:

- [Please Stop Using Local Storage](#)
- [Why LocalStorage is Vulnerable to XSS \(and cookies are too\)](#)
- [React Authentication: How to Store JWT in a Cookie](#)
- [Content Security Policy Reference](#)
- [SameSite cookies explained](#)
- [The Ultimate Guide to handling JWTs on frontend clients \(GraphQL\)](#)

## Other Advanced GraphQL Topics

After mastering some GraphQL essentials, you may be interested in reading about more nuanced topics such as how to [leverage nullability in a schema](#) and different approaches to [handling errors for GraphQL APIs](#). If you'd like to learn more about what the `info` parameter does in a resolver function, then it will be well worth 20 minutes of your time to watch the [Resolve Info Deep Dive talk](#) by William Lyon from GraphQL Summit 2019.

After you've spent some time working with monolithic GraphQL APIs, you may also be interested in learning about [Apollo Federation](#), which provides a declarative approach to building distributed GraphQL architectures.

Lastly, to learn more about how GraphQL execution works, watch the [GraphQL Under the Hood talk](#) talk by Eric Baer from GraphQL Summit 2017, as well as [this post on query lexing and parsing](#) and [this post on query validation](#) (both by Christian Joudrey).

## Public GraphQL APIs

For some GraphQL schema inspiration, you can check out the following public GraphQL APIs:

- [SWAPI GraphQL Wrapper](#)
- [The Rick and Morty API](#)
- [SpaceX Land API](#)
- [Countries GraphQL API](#)
- [Shopify - GraphQL Admin API reference](#)
- [GitHub GraphQL API](#)

## More Content from the Author

Finally, if you enjoyed this book, you can find conference talks and additional books by the author at these links:

- [Your First Federated Schema with Apollo Server](#)
- [How to Auth: Secure a GraphQL API with Confidence](#)
- [Using Subscriptions with Your Federated Data](#)
- [Advanced GraphQL with Apollo & React](#)

# About the Author

Mandi Wise has worked in developer education for over five years with a focus on full-stack web application development using JavaScript and GraphQL. During that time, she wrote curricula for and instructed numerous full-time and part-time courses and also developed software to support program delivery. She currently works as a Solutions Architect for Apollo Graph Inc.

Mandi first discovered her love for building things for the web 20 years ago while spending countless hours handcrafting beautiful table-based websites with spacer GIFs on GeoCities. She currently shares that enduring passion for the web by speaking at numerous developer conferences and meet-ups and actively volunteering in her local tech community.

# **Changelog**

## **v1.0.0**

- Initial release